Microware®

OS-9

REAL-TIME OPERATING SYSTEM

PRIMER

Mark A. Heilpern

The OS-9 Primer

By Mark A. Heilpern

MICROWARE SYSTEMS CORPORATION
Des Moines

The OS-9 Primer
First Edition

Copyright © Microware Systems Corporation 1994
All rights reserved.

Printed in the United States of America.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise without the express and prior permission of the publisher.

Microware Systems Corporation, 1900 NW 114th Street
Des Moines, Iowa 50325-7077

The information contained herein is believed to be accurate as of the date of publication; however, Microware will not be liable for any damages, including indirect or consequential, from use of the OS-9 operating system, Microware-provided software or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

Microware, OS-9, OS-9000 and PCBridge are registered trademarks of Microware Systems Corporation. FasTrak and Ultra C are trademarks of Microware Systems Corporation. UNIX is a trademark of UNIX System Laboratories. Pentium is a trademark of Intel Corporation. PowerPC is a trademark of IBM Corporation. All other brand or product names are trademarks or registered trademarks of their respective holders.

Cover design by Holtz Wimer Design Studios
Author's Photograph by Norm Stevens

ISBN: 0-918035-04-X
MSC Product Code: PRI68RA68MO

Acknowledgments

I would like to thank my wife, Vicky, for allowing me time to work on this project along with my many other activities. Also, thanks to Sammie and Carly, who had to go on many dog walks without me while this text was being written.

There are a lot of other people who deserve thanks for helping me put this book together. I hope I get to everyone, but please forgive me if your name should appear here and it doesn't. Thanks to Ken Kaplan for encouraging me to write this book and agreeing to publish it. Steve Johnson for his ideas as to what the book should include. Deb Fry and Steve Simpson for editing and formatting, and in alphabetical order: Joe Briggs, Drew Crane, Larry Crane, Angela Hefner, Daryl Jarman, R. Allan Kirkhart, Scott McGee, Boisy Pitre and Pam Weber for checking the technical aspects, as well as everyone else who has read through the manuscript at one time or another prior to going to press.

Introduction 1

How to Use This Book 3

1.1 Beginning 3

1.2 OS Items to Know About 4

1.3 Programming Issues 4

Planning an OS-9 Project 7

2.1 Development Environments 7

2.1.1 Resident Development 7
2.1.2 Personal Computer Hosted Development 8
2.1.3 FasTrak 9

2.2 Which Kernel to Use 10

2.2.1 Standard Kernel 10
2.2.2 Atomic Kernel 10

2.3 Memory Model 11

2.3.1 First Fit Allocation 11
2.3.2 Buddy Allocation 11

Getting Started with the Resident OS-9 System 13

3.1 Logging into the OS-9 System 13

3.1.1 Waking up the Time-Sharing Monitor 13

3.2 The Command Interface 14

3.2.1 Entering Commands 15

3.3 Environment Variables 19

3.3.1 The .login File 21

Moving About the OS-9 Shell 25

4.1 Command Line Modifiers 25

4.1.1 Redirection of I/O 26
4.1.2 Background Tasks 29
4.1.3 Command Separation 30
4.1.4 Command Piping 32
4.1.5 Process Priority 33
4.1.6 Stack Space 34
4.1.7 Wild Card Characters 36
4.1.8 Command Aliasing 36

4.2 Built-in Commands 37

4.2.1 ex 37
4.2.2 kill 37
4.2.3 logout 37
4.2.4 profile 38
4.2.5 setpr 38
4.2.6 w 38
4.2.7 wait 38

OS-9 Memory Modules 39

5.1 OS-9 Building Blocks 39
5.1.1 Module Header 40
5.1.2 Module Body 46
5.1.3 Module CRC 46
5.2 Other Standard Module Oriented Utilities 46
5.2.1 load 46
5.2.2 save 47
5.2.3 ident 47
5.2.4 fixmod 47
5.2.5 moded 48
5.2.6 link 48
5.2.7 unlink 48

OS-9 Devices 51

6.1 The Components of OS-9 51
6.1.1 The Kernel 51
6.1.2 User Applications and Utilities 51
6.1.3 init and clock Modules 52
6.1.4 Trap Modules 52
6.2 Devices and their Drivers 54
6.2.1 File Managers 54
6.2.2 Device Drivers 55
6.2.3 Device Descriptors 55
6.3 Accessing Devices 56

Paths and Processes 57

7.1 Path Descriptors 57
7.2 Standard Paths 58
7.3 Basic C Calls for Paths 58
7.3.1 error_code _os_open(char *name, u_int32 mode, path_id *path) 58
7.3.2 error_code _os_create(char *name, u_int32 mode, path_id *path, u_int32 perm, ...) 59
7.3.3 error_code _os_dup(path_id orig, path_id *copy) 60
7.3.4 error_code _os_close(path_id) 60

7.3.5 error_code _os_read(path_id path, void *buffer, u_int32 *count), _os_readln() 60
7.3.6 error_code _os_write(path_id path, void *buffer, u_int32 *count), _os_writeln() 61
7.4 Path Manipulation 61
7.5 Processes 63
7.5.1 The func Parameter 64
7.5.2 Priority 65
7.5.3 Path Count 65
7.5.4 Module Name 66
7.5.5 Argument Vector 66
7.5.6 Environment Pointer 67
7.5.7 Stack Override 67
7.6 error_code _os_wait(process_id *pid, status_code *stat) 67

Interprocess Communication — Data Transfer Methods 69

8.1 Pipes 69
8.1.1 Pipe Types 69
8.1.2 C Subroutine Calls for Pipes 71
8.1.3 Implementing a Timeout for Input with the Special Pipe (Path) Calls 73
8.2 Potential Problems with Pipes 74
8.2.1 Writing to a Full Pipe 74
8.2.2 Reading from an Empty Pipe 75
8.2.3 Creating a Pipe whose Name is Taken 75
8.2.4 Opening a Pipe that Does Not Exist 75
8.2.5 Closing Non-Empty Pipes 75
8.3 Pipe Sizes 76
8.4 Data Modules 76
8.4.1 Shared Memory 76
8.4.2 Creating a Data Module 77
8.4.3 Linking to a Data Module 79
8.4.4 Loading a Data Module from Disk 80
8.4.5 Saving a Data Module to Disk 81
8.4.6 Fixing the Module CRC 82
8.4.7 Unlinking from a Module 82
8.4.8 Crash Recovery using Data Modules 83

Interprocess Communication — Signals 85

9.1 Signals 85
9.1.1 Signal Intercept Routine 86
9.1.2 Sending a Signal to a Process 87
9.1.3 Masking Signals 89
9.1.4 Broadcast Signal Sending 91

9.2 Alarms 91
9.2.1 Alarms for a Certain Date and/or Time 92
9.2.2 Alarms to be Sent After Some Time Period has Elapsed 93
9.2.3 Cyclic (Repetitive) Alarms 96
9.2.4 Removing Alarms from the System 96

Interprocess Communication — Semaphores 97

10.1 Semaphores 97
10.1.1 Creating the Semaphore 97
10.1.2 Competing for the Semaphore 98
10.1.3 Releasing the Semaphore 98
10.1.4 Removing the Semaphore from the System 98
10.1.5 Sample Code Using a Semaphore 98
10.2 Events 101
10.2.1 Creating a Non-Existing Event 101
10.2.2 Linking to an Event Already Created 101
10.2.3 Waiting for an Event to Occur 101
10.2.4 Signaling that an Event has Occurred 102
10.2.5 Reading the Value of an Event 103
10.2.6 Setting the Value of an Event 103
10.2.7 Pulsing an Event 104
10.2.8 Accessing an Event on Relative Terms 106
10.2.9 Unlinking from an Event 106
10.2.10Removing an Event from the System 106

Process Scheduling 107

11.1 Active Queue 108
11.2 Sleep Queue 111
11.3 Wait Queue 111
11.4 Event Queue 112
11.5 Semaphore Queue 112
11.6 Device I/O Queue 112
11.7 Process Preemption 113

Combining Assembly Language with C Routines 115

12.1 Hardware Information 115
12.2 Subroutine Parameters 116
12.3 Register Class Variables 117

12.4 Stack Frame Pointer 117
12.5 Global Variable Pointer 120
12.6 Placing Assembly Directly within a C Code Module 121

Locating and Executing Code External to the Main Program Module 123

13.1 Subroutine Modules 123
13.1.1 Coding a Subroutine Module 124
13.1.2 The Index Table 126
13.1.3 The Functions of a Simple Subroutine Module 127
13.1.4 The Program that Accesses a Subroutine Module 128
13.2 Trap Modules 130
13.2.1 Coding the Trap Module 131
13.2.2 The Program that Accesses a Trap Module 134

Exception Handling through C 135

14.1 Types of Exceptions 135
14.2 Telling the System About Your Exception Handler 137
14.3 Handling the Error Without Terminating Your Program 140

Interrupt Service Routines 143

15.1 Interrupts vs. Exceptions 143
15.2 The Standard 68k Interrupt Mechanism 144
15.3 OS-9's Unique Approach to Interrupt Handlers 144
15.3.1 Finding Out What Interrupts are Currently Installed 146
15.3.2 Installing a System Polled Interrupt Service Routine 147
15.3.3 Installing a "Fast" Interrupt Service Routine 147
15.3.4 Removing an Interrupt Service Routine from the System 147
15.4 A Simple Program to Install an Interrupt Routine 148
15.4.1 Masking Interrupts 153

OS-9 System Global Variables 155

16.1 More on Process Scheduling 155
16.1.1 Altering the Effects of Aging 155
16.1.2 Suspending Processes 158
16.1.3 Ticks Per Time Slice 158
16.2 Measuring Time 158
16.2.1 Unitless Measurement 158

16.2.2 Gaining Meaning From this Measurement 159
16.3 Determining Processor Types 159
16.3.1 CPU Type 159
16.3.2 FPU Type 160
16.4 Other Globals 160
16.4.1 System State Preemption 160
16.4.2 Pointer to the Current Process Descriptor 160

Memory Management on the OS-9 System 163
17.1 Memory List 163
17.2 Methods of Memory Allocation 164
17.3 Memory for Processes 165
17.3.1 malloc() 166
17.3.2 Lower Level Allocation 167

Issues for Custom System Design 169
18.1 Items Required to Boot OS-9 169
18.1.1 Init Module 169
18.2 Booting 175
18.2.1 Booting From Disk 175
18.2.2 Booting From ROM 177

The μMACS Editor 179

Differences Between MShell and Shell 183

FIGURE 1	Logging into the OS-9 System	14
FIGURE 2	The dir Command	15
FIGURE 3	Extended Directory Information	16
FIGURE 4	Obtaining Usage Help for a Command	17
FIGURE 5	Changing Directories	18
FIGURE 6	Environment Variable Commands	20
FIGURE 7	Compiling and Running envtest	23
FIGURE 8	Execution Directory Specifications	24
FIGURE 9	Redirection of Standard Output	26
FIGURE 10	Redirection Appending and Overwriting	27
FIGURE 11	Redirection of Standard Error	28
FIGURE 12	The procs Command	30
FIGURE 13	Executing Commands Sequentiall	31
FIGURE 14	Command Piping	33
FIGURE 15	Giving Additional Stack Space to a Process	35
FIGURE 16	Wildcard Usage	36
FIGURE 17	Module Format	40
FIGURE 18	Extended Module Directory Information	41
FIGURE 19	A High Level View of OS-9	53
FIGURE 20	The Pipe Directory	71
FIGURE 21	Location of Code within a Subroutine Module	125
FIGURE 22	Irqs Command Output	146

TABLE 1	Environment Definitions	21
TABLE 2	Attribute Values and their Meanings	42
TABLE 3	Module Types and their Definitions	45
TABLE 4	Defined Signal Values	86
TABLE 5	Process Scheduling: A Simple Case	109
TABLE 6	Register Convention of Microware's C Compiler	116
TABLE 7	Hardware Exception Types	136
TABLE 8	Common Command Sequences for the µMACS Editor	180

Introduction

This text was not designed to be a replacement for the Microware manual set, but a companion to be used along with the manuals. When sitting down to write this book, I drew a lot of information from the 5-day intensive OS-9 training course I have been teaching for the past 16 months, including much of that course's material. Also, within this text I tried to answer many of the questions that my students have asked in class.

Before reading this text, you should have some experience with an operating system, be it UNIX, DOS or just about any other. This book is not intended to contain a built-in OS course, rather to explain how the OS-9 system can be used in an application. All of the code examples are written in C (unless assembly is the only option), the current "language of choice" among the vast majority of real-time software engineers. C is not the only language supported under OS-9; it is merely the most popular. Currently, support also exists for Ada, Fortran, Forth, Modula-2 and Pascal, either directly through Microware or through third party vendors. In addition, C++ may soon be supported by the Ultra C compiler.

Recently, a fair number of system users have been making the switch to OS-9000, perhaps since that operating system is designed with portability to other processors in mind. Currently, OS-9000 is available for the 386, 486, Pentium, and PowerPC microprocessors.

On the surface, OS-9 and OS-9000 look and feel very similar, leading to a fair amount of ease in porting applications from one system to the other, but by no means are both systems identical. This book concentrates on OS-9, Version 3.0 (which is very similar at the user state code level to Version 2.4), but occasionally I make a reference to how things are done under OS-9000 to provide a small degree of contrast.

Although I focus on OS-9, this text may serve as a good learning tool to those pursuing education in OS-9000 until a book on this subject becomes available.

All of my code examples have been written to compile under the Ultra C compiler running in its default mode of extended ANSI. The function names and parameters for standard operations under this compiler's libraries are very similar — and in most cases identical — to the names and parameters required for OS-9000. This of course improves the prospects of compatibility and portability of user state code on these operating systems.

Enjoy...

Mark A. Heilpern

CHAPTER 1

How to Use This Book

There are many chapters in this book and it may not be immediately apparent which ones should be read by whom. In each chapter there is the possibility of the need to draw upon information that has previously been presented, but these occurrences are not very prevalent. This chapter provides a simple road map to each of the book's remaining chapters.

1.1 Beginning

Before sitting down to write your application, you must have some sort of OS-9 development system. Chapter 2 highlights the various options available for OS-9 development systems.

The third and fourth chapters present topics mainly relevant to people developing on the resident OS-9 operating system (people not doing cross-development on systems such as UNIX or DOS).

Chapter 3 begins by bringing you through the login process and describes some of the very basic utilities including how to obtain a listing of the files in a directory and how to change directories. Also discussed are environment variables and the issue of data directories versus execution directories.

Chapter 4 presents an overview of the OS-9 shell, the most important utility used by anyone developing OS-9-based applications using OS-9 as an operating system.

1.2 OS Items to Know About

Beginning with chapter 5, the book discusses some general items about the operating system and its components. Very few programming issues are discussed here.

Chapter 5 presents the theory of memory modules, the building blocks that make OS-9 an easy system to configure. The layout of these modules are described and some resident commands pertaining to memory modules are presented toward the end of the chapter. (Although the commands presented are resident, the cross-platform development kits have equivalent commands for performing similar actions.)

Chapter 6 discusses the device I/O subsystem OS-9 makes available. This chapter does not teach how to make device drivers for OS-9, but describes how they relate with other components on the system.

Chapter 7 details information concerning I/O paths on the OS-9 system, including descriptions of some routines available for creating and manipulating paths. Also discussed in this chapter are methods of creating child tasks (processes).

1.3 Programming Issues

The next chapters contain information used in building multiple-process applications to run using the services provided by the OS-9 kernel.

Chapter 8 kicks the section off with information relating to the transfer of data among processes, one class of interprocess communications. Specifically, pipes and data modules are presented in this chapter.

Chapter 9 continues the discussion of interprocess communications with information concerning signals. Signals and alarms are both presented as methods of receiving software-controlled interrupts within an application.

Chapter 10 wraps up the IPC discussion by describing semaphores and events, tools used in resource management.

Chapter 11 takes a small break from programming issues to discuss how process scheduling takes place on the OS-9 system. The issues of priority, aging, and preemption are prevalent.

Chapter 12 discusses register convention used by the C compiler as well as methods for interfacing C with assembly. This chapter sets the stage for the chapters that follow.

Chapter 13 presents the methods of building and using shared libraries. These tools are useful for building code that many different programs will use and are also useful for implementing "overlays" on memory-constrained, disk-based systems.

Chapter 14 shows you how to write code to catch hardware exceptions. This is an art that can make the difference whether a process runs or terminates after causing a bus error or division-by-zero error, among others.

Chapter 15 presents interrupt service routines outside the scope of a regular device driver. Also in this chapter is a full program to install an intercept service routine on any system that already has interrupts by "borrowing" the other routine's interrupt vector.

Chapter 16 shows how to access the system's global variables. Some of these globals are read-only and present useful information about the system. Other globals may be modified by privileged processes; these modifications alter the way the system will run.

Chapter 17 presents the tools available for memory requests and discusses the various types of memory areas and allocation available under OS-9.

Finally, Chapter 18 presents a discussion of the many issues surrounding design of an OS-9 embedded system.

CHAPTER 2

Planning an OS-9 Project

Several options for how to develop an OS-9 project are available. This chapter presents the major options and their relative strengths and weaknesses, allowing you to make an informed decision on a development environment (assuming, of course, that you currently do not have one)!

2.1 Development Environments

2.1.1 Resident Development

One option for where the development of your project will take place is resident to the OS-9 operating system. This system is not just a real-time kernel with I/O support, but a full operating system complete with all of the tools and utilities necessary to get the development job done. Developing code directly under OS-9 has the benefit of not requiring additional computer systems beyond what is needed to perform your final task.

OS-9 is a very modular operating system and many of the modules available do not maintain a dependance on others, thus when the development task is completed you simply remove those modules you no longer use. Examples of modules used during development but not necessarily after-

wards include those needed for disk drive access, network support, debug support and many others. Of course, if your application requires some of these items, they may be left on the system.

2.1.2 Personal Computer Hosted Development

A product called PCBridge is available to allow development of an OS-9 system on a standard IBM compatible computer running DOS or an equivalent. PCBridge provides a complete cross compiler allowing OS-9 modules to be created while on the PC. Along with the cross compiler, PCBridge provides the definition files needed to compile objects as well as the standard OS-9 libraries. Also included is the Source Level Debugger, providing a means of perfecting your application once it has been compiled.

PCBridge communicates with OS-9, when necessary, through a serial interface between the PC and OS-9 systems. This, of course, implies that communications are not very quick. However, communication between systems is kept to a minimum, as the only times the two must pass data are when 1) you have initiated a "dumb terminal" connection between systems and 2) you decide to download (or upload) modules and files to the OS-9 system. When developing, all editing and compilation is performed on the PC with no interaction between it and OS-9. When you wish to execute an application, you must first download all related modules to OS-9, then establish a shell connection and type the commands necessary to begin execution. If you wish to perform symbolic source debugging on the object, you must also download all source files (to a disk device or RAM disk) and additional symbol modules as well as the debugger.

PCBridge's advantage over resident development is that you may use tools provided through third party vendors, such as your preferred editor. Under resident development, it is very difficult to find the wide array of tools available for the PC.

At the time of the writing, a Windows version of FasTrak (see below) is about to enter the beta stage. This version of FasTrak is essentially the same as the UNIX version, save the workbench (which is already built into the Windows Program Manager) and the automated error repair facility.

2.1.3 FasTrak

FasTrak, Microware's flagship of development products, is a complete cross-development environment. It is a UNIX X-Windows or PC MS-Windows based package. Rather than being simply a remote edit-compile-debug set of tools, this product provides an integrated environment housing all functions you need in developing applications for OS-9. FasTrak's functionality:

1. Uses any TCP/IP connection to communicate with OS-9. (Ethernet is preferred, but a SLIP [serial line internet protocol] connection will suffice.)
2. Includes a "workbench" to connect all development tools through a mini-window system.
3. Includes an editor front end to allow your text-based editor of choice (such as vi or µMACS) to run inside a graphics window provided with on-screen button functionality.
4. Includes automatic code repair facilities which are activated after compiling a source file containing errors.
5. Includes a makefile generator to create a list of source file dependencies to header files, allowing for automated building of applications.
6. Includes a source level debugger which does not require advance downloading of any modules, but downloads (automatically) only the module to be executed and uses host resources for all other items.
7. Includes an intelligent target system window which acts as a dumb terminal and more. This target window allows you to seamlessly execute commands which reside on the host as if they were resident on the OS-9 system (with automatic background downloading when necessary). It also allows you to profile your applications, showing you where the most CPU time is spent across the many modules in your multi-tasking application.

If I go on about FasTrak any longer, you'll think I'm trying to sell you something. Suffice it to say that this product is most comprehensive in what it provides.

2.2 Which Kernel to Use

OS-9 now provides two different kernels for each processor supported: a standard kernel and an "atomic" kernel, each with two different memory allocation schemes: "first-fit" and "buddy."

2.2.1 Standard Kernel

The standard kernel closely resembles the kernel you may be used to from version 2.4 of the OS, differing mainly in that it is now optimized for each processor it runs on, and part of the I/O subsystem has been made a separate unit from the kernel.

This larger model of kernels is required for any serious development work since it has a complete set of development facilities.

2.2.2 Atomic Kernel

The atomic kernel is smaller in size and in its data requirements, quicker in execution time and lacking in some development facilities. This smaller version of the kernel was designed for use in embedded systems which have an emphasis on execution time. The atomic kernel is smaller in size because support for several items has been removed, including C source level debug support, MMU (memory management unit) support, system table expansion and user differentiation. These items are rarely found on final solutions and will probably not be missed.

During the development phase of a project, I recommend using the standard kernel. Then, when you are ready to embed everything, you may wish to switch to the atomic kernel to increase execution speed and decrease ROM

requirements. In nearly every case, the only side effects you will notice are a decrease in total code size and an increase in kernel execution speed.

2.3 Memory Model

For each CPU supported there are four kernels available: atomic with standard allocation, atomic with buddy allocation, standard with standard allocation, and standard with buddy allocation.

2.3.1 First Fit Allocation

An additional memory allocation method has been made available with this version of OS-9. Until now, the memory allocation scheme used was "first fit," meaning that as soon as an area of memory large enough to accommodate the memory requirement is found, it is allocated from that area. Although this method sounds as if it may be inefficient with the dolling out of memory, it does a reasonable job most of the time. The downside of first fit allocation lies in its worst case response time. This basically means that the more fragmented the memory space is, the longer it takes to accommodate a memory request. Because of the number of parameters involved in determining the amount of time it will take to grant memory, this method is often called non-deterministic. Of course, everything a computer performs (today) is deterministic but the degree of difficulty in determining timings for this method of allocation is very high.

2.3.2 Buddy Allocation

This form of memory allocation, new to OS-9, gets its name from the fact that memory is only granted in size powers of two. Unless more than half of available memory is granted at once, each memory request splits a fragment of memory into in halves, or buddies, perhaps giving one half to the requestor and placing the "buddy" half into the memory-

free pool. If the half that might be given to the requestor is twice the size of the memory request or greater, it will be further split.

For example, lets assume there is currently one megabyte of free memory, all in one chunk. Also assume a request is made for one kilobyte. After the request, the process asking for memory would end up with one K, and the free memory on the system would be fragmented into the following chunks: 512K, 256K, 128K, 64K, 32K, 16K, 8K, 4K, 2K, and 1K.

This form of allocation obviously leads to greatly fragmented systems but has a fairly constant timing associated with each allocation request. It does not take fragmentation in regard.

If you are on a system which makes many separate memory allocations and returns, and if execution speed is an important factor to your system, I would recommend using the buddy allocator. Otherwise the standard allocator should suffice.

CHAPTER 3

Getting Started with the Resident OS-9 System

This chapter will teach you how to login to the OS-9 system and will present you with some basic commands to get you moving.

3.1 Logging into the OS-9 System

When the OS-9 system is started up in a multi-user environment, you must get through minimal security before gaining access to the system. Typically this is a matter of getting the attention of the time-sharing monitor, tsmon, which oversees login activity. Once tsmon is aware of your intentions, it will start up the login program which will ask for your user name and, if applicable, password. If the information you supply is valid for your system, you will have successfully logged into OS-9.

3.1.1 Waking up the Time-Sharing Monitor

To wake up tsmon you must hit the return key on your terminal. Upon seeing this, tsmon starts up login and you will be prompted to enter a user name. You are given three attempts at a correct user name and corresponding pass-

word. After three failures, login will give up and tsmon will require you to wake it again.

Assuming you have successfully logged in, the system will display the current MOTD (for "message of the day") if one is available, do some minimal housekeeping and let you take over.

FIGURE 1 Logging into the OS-9 System

```
OS-9/68040 V3.0  Iggy_MVME167 - 68040  94/11/07 16:38:44
User name?: heilpern
Password:
Process #97 logged on   94/11/07 16:38:52
Welcome!
$
```

➤ When entering a password, you will not see your text echoed back to the screen. This is to keep casual observers from viewing your password.

3.2 The Command Interface

Now that you are logged in, you are prompted to enter a command. Figure 1 above has this prompt displayed as super but on your system it may be quite different. As we will see later, the prompt is a changeable string of text although, by default, it is simply a dollar sign ($).

A special program is running to give you this prompt, most likely a program called shell. In the eyes of the computer, shell is simply a program. Since the OS-9 system is multi-tasking, many users on the system can run shell simultaneously

3.2.1 Entering Commands

3.2.1.1 dir

To have the system perform a task, you must tell the shell what you want. As long as the shell can figure out your input, the task will be performed. For example, suppose you would like to get a listing of the files in your current directory. The dir utility is designed for that purpose and it may be executed by typing the command dir at the command prompt, as seen in Figure 2 below.

➤ Note: It is assumed the reader has a knowledge of hierarchial file systems. If this is not the case, refer to Appendix A for a discussion concerning them.

FIGURE 2 The dir Command

```
$ dir
                        Directory of . 16:45:27
RELS          defines.h     disp.c        edlin.c       fsdb.h
functions.l   lex.c         lex.h         main.c        makefile
parse.c       printlib.c
$
```

Among the output of dir are the names of all files in the current directory. Of course, the names of files are not of great use unless you can determine other information, such as the size of each file or whether the file is a normal file or directory. To gather extended information during a directory listing you may still use the dir command, but in a slightly different manner, as seen in Figure 3.

FIGURE 3 Extended Directory Information

```
$ dir -e
                          Directory of . 16:54:00
 Owner  Last modified   Attributes  Sector  Bytecount  Name
-------  ---------------  ----------  -------  ----------  -----
  0.142  94/05/30 0925   d-ewrewr    8A1F8   256         RELS
  1.142  92/07/24 1222   ------wr    897C0   1103        defines.h
  1.142  92/07/24 1222   ------wr    8EC10   7040        disp.c
  1.142  92/07/24 1222   ------wr    897C8   611         edlin.c
  1.142  92/07/24 1222   ------wr    89828   516         fsdb.h
  0.151  92/06/02 1452   ----r-wr    8ED10   24093       functions.l
  1.142  92/07/24 1222   ------wr    8A108   1407        lex.c
  1.142  92/07/24 1222   ------wr    8A110   507         lex.h
  1.142  92/07/24 1222   ------wr    8EC20   5558        main.c
  1.142  92/07/24 1224   ------wr    8A1B8   518         makefile
  1.142  92/07/24 1222   ------wr    8EC40   5859        parse.c
  1.142  92/07/24 1222   ------wr    8A1F0   1546        printlib.c
$
```

➤ One of the file names shown above are in capital letters, though OS-9 file names may be either capital or lower case. Their appearance will be in the case the file was created with, however you may refer to the file in any case (or combination of case). Thus, you may not have two files in one directory called FILE and file. The convention used is that all directories are wholly upper case and other files are not.

With the addition of the -e flag the dir command displays extended information. But, what should you do if you forget about that flag or would like to know what other flags are available for this command? Of course, you could read the manual but some unknown force often prevents software engineers from doing so. To combat this mystery of the trade, Microware has placed built-in usage information in nearly all utilities. That information can be seen with the universal flag, -?, as shown in Figure 4.

FIGURE 4 Obtaining Usage Help for a Command

```
$ dir -?
Syntax: dir [<opts>] {<dir names> [<opts>]}
Function: display directory contents
Options:
        -a          show all files
        -d          show directories with a slash
        -e          extended dir listing
        -n          treat dirs like files
        -r          recursive dir listings
        -s          unsorted dir listing
        -u          unformatted listing
        -x          directory is execution dir
        -z          get list of dir names from standard input
        -z=<path>   get list of dir names from <path>
$
```

3.2.1.2 pd, chd

Ok, now you know how to get a listing of your current directory. But, where on the disk are these files? The command pd will print the absolute location of your current directory.

The command chd is used to change your current directory. It is used with one parameter: the name of the directory you wish to change to. This location may either be absolute (referenced from the root directory of your disk) or relative (referenced from your current directory). There is a special directory called ".." (pronounced "dot dot") which is used to go one directory level higher. Figure 5 shows this being used. Also in Figure 5 you see three dots being used to refer to two levels up.

FIGURE 5 Changing Directories

```
$ pd
/h0/LEVEL1/LEVEL2/LEVEL3
$ chd ..
$ pd
/h0/LEVEL1/LEVEL2
$ chd ...
$ pd
/h0
$ chd LEVEL1/LEVEL2/LEVEL3
$ pd
/h0/LEVEL1/LEVEL2/LEVEL3
$ chd .....
$ pd
/h0
$
```

You also see a reference of many dots. The dotted directory macro will bring you up one level for each dot excluding the first. (The first dot refers to your current directory.) There is no reason for concern of too many dots; they will never bring you beyond the root directory of your disk.

3.2.1.3 Root Directory of a Disk

While in the root directory of the disk, as shown in Figure 5, the name of the directory is not "/" as would be the case on a UNIX system. Instead, under this OS-9 system, the directory is "/dd". This identifies the name of the disk device we are currently using. An OS-9 system may have many disk devices; each will have a different name to identify it, always beginning with a slash. Thus, OS-9 does not have a single root directory as UNIX, but several as DOS.

➤ Disk names refer to the hardware, not the physical media. Thus, for a removable media drive such as a floppy drive, the name would be constant without regard for what floppy is in the drive at any point in time. However, there could be different names referring to a single piece of hardware, in cases where either an alias to the unit is used, or perhaps different disks have different formats, i.e., OS-9 Universal format vs.

DOS format. Some common examples are: /dd and /h0 which generally point to the same device if the hard drive called h0 is the boot drive (dd is, in this case, an alias to h0), and /d0 and /pcd0 which generally refer to the same floppy hardware but different formats. (In the latter, d0 refers to a driver/file manager for OS-9 Universal format and pcd0 refers to a disk formatted in the standard 640k DOS format.)

➤ I've just insinuated that the dd device is the disk used to boot the system; this is not necessarily correct, although it is usually the case. Dd is the default disk that will be searched for information needed by Microware utilities. It may be any device on the system and it is not even required to be present.

3.3 Environment Variables

Each process under OS-9 maintains an area of memory called the process environment. This area of memory is arranged such that the system may identify the names of variables as well as their contents. (All of these variables are treated as null terminated character strings.) When one process creates another, the environment of the parent (creator) process is copied to that of the child (created) process. For example, when you type the command dir, the current environment of the shell is copied to the dir process. It is important to realize that the child process only has a copy of the parent's environment since any changes made to the environment by the child will not affect the parent process.

The printenv command allows you to view the current environment of your shell. You may add new variables or change existing ones with the command setenv and you may remove a variable with the command unsetenv.

FIGURE 6 Environment Variable Commands

```
$ printenv
PORT=/term
HOME=/h0/USR/HEILPERN
SHELL=shell
USER=heilpern
PATH=/h0/cmds
TERM=vt100

$ setenv MYNAME Mark Heilpern
setenv requires two arguments
$ setenv MYNAME "Mark Heilpern"

$ printenv
PORT=/term
HOME=/h0/USR/HEILPERN
SHELL=shell
USER=heilpern
PATH=/h0/cmds
TERM=vt100
MYNAME=Mark Heilpern
$
```

Notice that in Figure 6 the first attempt at using setenv resulted in an error message, stating that the command requires exactly two arguments. The command as specified contains three: NAME, Mark and Heilpern. To get beyond this limitation, the desired contents of the variable are placed within quotes effectively turning two words separated by a space into one argument.

The environment variables listed by the printenv command in Figure 6 are some of the most common variables (except for NAME). Their specific meanings are listed in Table 1.

TABLE 1 Environment Definitions

Name	Meaning
PORT	The name of the serial device you logged in with.
HOME	The name of your home directory.
SHELL	The name of your command interface program.
USER	The name you logged in with.
PATH	A list of directories, each separated with a colon, allowing the system to find the locations of commands you wish to execute.
TERM	The type of terminal hardware you are using.
PROMPT	The text used by the shell to ask you for the next command.

In Section 3.2.1.2 it was mentioned that the chd command needs one parameter: the name of the directory to change to. Actually, the chd command may be used without that parameter if you wish to change to your home directory. To do this, however, your HOME environment variable must be set to your home directory. If you are logging in through normal means (tsmon/login), this will be set automatically, along with PORT, USER and SHELL. Other environment variables are not set for you by the system, but you may have them set upon login if you wish.

3.3.1 The .login File

Users typically have a normal setup they like to log into, including properly set PATH, TERM and PROMPT environment variables (three things the system does not perform for you). To automate these assignments, the shell provides

for a script file to be executed upon logging into the system. This file is called ".login" and must be placed in the user's HOME directory. The contents of this file are commands that could be typed at the shell prompt. An example of a .login file is shown in Listing 1.

Listing 1. The Contents of a .login File

```
* an asterisk on the beginning of a line
* is a comment!

* set our environment variables as
* desired
setenv TERM vt100
setenv PATH /dd/CMDS:/dd/CMDS/MARK
setenv PROMPT $

*display a greeting
echo "Hello, Mark."
```

➤ When a dir listing is gathered, the .login file will not appear! This is because files whose name begins with a period are treated as semi-hidden files. They do not appear in a directory listing unless the -a flag is included on the command line. That flag is used to make *all* files appear.

3.3.1.1 Source Code for Getting Environment Variables

The program envtest shown in Listing 2 contains the code necessary to obtain the contents of an environment variable. This example looks for a variable called NAME and, if found, displays a short greeting. If the variable is not found, a suitable error message is presented.

Listing 2. Retreiving Environment Variables Within Your Program

```
/**** File: evntest.c ****/
#include <stdio.h>

main()
{

   char *theVariable;

   theVariable = (char*)getenv("NAME");

   if (theVariable==(char*)NULL)
```

```
    {
        fprintf(stderr,"I need the
            environment variable NAME set
            first!\n");
        exit(0);
    }

    printf("Hello, %s, I hope you are
            learning from this text!\n",
            theVariable);
    exit(0);
}
```

➤ If you need to learn how to input programs under a resident OS-9 system, consult Appendix A which is an introduction to the μMACS text editor.

This simple program is compiled with the C compiler, cc. Because the program is self-contained, the command line will be a simple cc envtest.c. See Figure 7.

FIGURE 7 Compiling and Running envtest

```
$ cc envtest.c
$ envtest
I need the environment variable NAME set first!
$ setenv NAME Reader
$ envtest
Hello, Reader, I hope you are learning from this text!
$
```

3.3.1.2 Execution Directory

Note that after the compilation, the program was executed by running the command envtest. However, when a listing of the current directory was gathered it was not there. This is because OS-9 maintains not one current directory, but two. The first, a data directory, is the current directory you may be used to. This is the directory to which most operations default. The second is called an execution directory, where commands are searched for at execution time. The C compiler places the final object in the execution directory and the system searches there prior to searching the PATH

environment variable directories when preparing to execute a command.

We've seen the pd command show the current data directory; this command also shows the current execution directory if the -x flag is used. Should it become necessary to change the current execution directory, chx is used similarly to chd. There is one difference between chd and chx: chx will not accept a missing parameter under any circumstances.

FIGURE 8 Execution Directory Specifications

```
$ dir -x
                Directory of . 17:42:32
envtest
$
```

The -x flag supported by pd is a common flag used by many programs that have filenames as parameters which default to relative reference from the data directory. Figure 8 shows the dir utility being used to get a listing of the files in my execution directory.

CHAPTER 4

Moving About the OS-9 Shell

This chapter will teach you many of the built-in features of the standard command interface for OS-9, the shell. Appendix B contains information on MShell, an enhanced command interface sold by Microware.

4.1 Command Line Modifiers

In Chapter 3 we saw a few examples of changing the way a command works by specifying optional command line arguments (flags), e.g., the dir utility could be told to display extended information by including the -e flag on the command line. Flags must be programmed into the utility to be recognized. The OS-9 shell also has methods of changing how a command will execute by including modifiers on the command input line. These modifiers are tokens that are interpreted by the shell and thus do not have to be written into the utility. In fact, under most circumstances, the program being run has no knowledge of the shell command modifiers having been present.

4.1.1 Redirection of I/O

The first set of command modifiers allows the user to change the origin of input as well as the destination for output of a process.

4.1.1.1 Modified Standard Output

The greater-than symbol (>) is the token used to modify the standard output path for a process. This may be used to redirect output to either a file or device. Figure 9 shows this modifier in use.

FIGURE 9 Redirection of Standard Output

```
$ dir >dir.out
$ list dir.out
                    Directory of . 08:09:44
dir.out      file1        file2
$ dir >dir.out
shell: can't open "dir.out" - Error #000:218
$
```

This usage shows the output of the dir utility being redirected into a file called dir.out which is created in the current data directory. Notice that there is *no* space (or other character) between the redirection symbol and the destination. This is as the shell requires it.

After executing the command, we use the list utility to view the contents of a file and the file contains what we would expect from the dir utility.

Notice that we tried to execute:

```
dir >dir.out
```

a second time and failed due to OS-9 error 000: 218. This error number indicates that we tried to create a file that already exists. OS-9 will not let you overwrite an existing

file by redirecting into it unless you are sure of what you are doing. To get past this we can place either a minus sign or plus sign between the redirection symbol and the first character of the destination. Using a minus sign indicates that we wish to replace the contents of the file with new information while a plus sign indicates that we wish to append to the contents of the file.

➤ For an on-line source of the meaning of Microware error numbers, try browsing through the file /dd/sys/errmsg. On some systems, the file may be found in the MWOS development structure, .../MWOS/OS9/SRC/SYS/ERRMSG/errmsg.

FIGURE 10 Redirection Appending and Overwriting

```
$ count dir.out
"dir.out" contains 2 lines
$ dir >+dir.out
$ count dir.out
"dir.out" contains 4 lines
$ dir >-dir.out
$ count dir.out
"dir.out" contains 2 lines
$
```

Rather than show the contents of the file, I've chosen to use OS-9's count utility to count the number of lines in the output file.

Rather than redirecting to a file, we could have redirected to any device. For example, if we wished to send the output of our command to a terminal connected to OS-9 device t1, we would have issued the command dir >/t1. Note the slash before the device name. As mentioned earlier, the slash identifies what follows by device name and, if supported by the particular device, a file name.

4.1.1.2 Modified Error Output

Commands that output error messages usually send these messages to a path other than standard output (stdout). Rather, error messages are generally sent to the error output

path, stderr. This can be observed in Figure 11. Also seen is the operator for error redirection, two greater-than symbols (>>).

FIGURE 11 Redirection of Standard Error

```
$ dir nothing >dir.err
dir: can't open "nothing". Error #000:216
$ dir nothing >>-dir.err
Error #000:216
$ list dir.err
dir: can't open "nothing". $
$
```

The first attempt at redirection of standard output does not help divert the error message dir: can't open "nothing". Error #000:216 from being displayed since this is error output and we only redirected standard output. The second attempt at redirecting error output works, sort of! Although the error text did get placed into the redirection file, there is still an error number being displayed. This is caused by the dir command exiting with an error code, error 000:216. This number is not printed by the dir utility, however. The shell is informed as to its child process' exit status (the error number) and informs the user via this message. The only way to avert this output would be to redirect standard error for the shell itself. (Or, rewrite the shell!)

4.1.1.3 Modified Standard Output and Error Output

When wanting to redirect both standard and error output paths, the shell makes things easier than the intuitive solution. Using intuition one would try a command such as:

```
dir >dir.out >>dir.out
```

In principal, the format of this example is ok as long as the stdout and stderr paths are different. Unfortunately, this command will always fail if issued as shown. The reason for this is the shell will, in this example, try to create the file dir.out twice. If it does not exist the first time, it will the

second, giving us an error we have seen previously, 000:218. However, the shell does give us an avenue of escape. The command:

```
$:dir >>>dir.out
```

will start the dir process with both standard output and error output going to the file dir.out.

4.1.1.4 Modified Standard Input

For commands that require further input than what is placed on the command line, standard input (stdin) can be redirected as well. The redirection of standard input token is the less-than symbol (<), as in:

```
cmd <file.in
```

This is most useful when repeatedly testing a program which requires user input, provided the test-case answers do not change.

4.1.1.5 Combining I/O Redirections

Just as standard output and error redirections may be combined, standard input may be combined with other redirections. For example, assume you would like to run some program on another terminal rather than yours, e.g., a shell on device /t2:

```
shell <>>>/t2
```

This command line will start up a shell with all I/O being redirected to the device called t2.

4.1.2 Background Tasks

Not only can the OS-9 system run several tasks simultaneously, but each user can have multiple tasks running as well. The command procs shows all processes being run by the user executing the command.

FIGURE 12 The procs Command

```
$ procs
 Id PId Grp.Usr  Prior  MemSiz Sig S   CPU Time   Age Module & I/O
 78  63  0.142   128   28.00k   0 w     1.62  0:16 shell <>>>term
116  78  0.142   128   32.00k   0 *     0.08  0:00 procs <>>>term
$ shell <>>>/t2 &
+111
$ procs
 Id PId Grp.Usr  Prior  MemSiz Sig S   CPU Time   Age Module & I/O
 78  63  0.142   128   28.00k   0 w     1.69  0:16 shell <>>>term
111  78  0.142   128   12.00k   0 s     0.11  0:00 shell <>>>t2
121  78  0.142   128   32.00k   0 *     0.11  0:00 procs <>>>term
$
```

4.1.2.1 The Ampersand Operator

In Figure 12 you see the procs command executed twice, once before a background operation is started and once after. Between these procs commands the new shell is started with all I/O redirected to the t2 terminal. Notice the "&" character at the end of the command line. Also notice that the shell on our terminal returns right away with the simple output +111. As seen in the output of the next command procs, 111 is the process id number assigned to the new shell process. Procs also shows that standard I/O for that process is going to device t2.

We have just started a command shell with all I/O redirected to another device and running in the background with respect to our main shell. Starting a new shell like this saves us from having to login on another port, yet still gives us simultaneous access to two shells.

4.1.3 Command Separation

When specifying commands on the input line, it is sometimes helpful to group several commands together rather than invoking each on separate lines.

4.1.3.1 Sequential Execution

The semi-colon (;) is used to separate commands when you wish each command to follow the previous in synchronous execution. That is, input of the form:

cmd1; cmd2; cmd3

specifies that first program cmd1 is to execute; when it completes, cmd2 will run; and when it completes, cmd3 will run. Requesting a list of sequential commands like this works to your advantage if you do not want commands toward the end of the list to execute if commands toward the beginning fail. The first command to fail aborts execution of all commands following. For example, assume there is no file called nothing in the current directory (see Figure 13). Notice that the first command, dir nothing was executed, but the second, dir was not.

FIGURE 13 Executing Commands Sequentially

```
$ dir ; dir nothing
                    Directory of . 08:41:30
dir.err      dir.out      file1        file2
dir: can't open "nothing". Error #000:216
$
$ dir nothing ; dir
dir: can't open "nothing". Error #000:216
$
```

➤ In this figure's first combination command line, dir; dir nothing, both commands execute with the second running until an error terminates it. The second combination command line encounters an error with the first command and the second command on the line does not even begin to run.

4.1.3.2 Concurrent Execution

Besides providing for background operation, the ampersand is available to separate commands you wish to execute asynchronously. Input of the form:

cmd1 & cmd2 & cmd3

indicates that you wish cmd1, cmd2 and cmd3 to begin running at (as close as possible to) the same time. Errors caused by one of these commands will have no bearing on the execution of others on the input line.

4.1.3.3 Command Grouping

Using parenthesis, commands entered on the same input line may be grouped together. For example, assume you would like to execute the dir command followed by the procs command, giving each of these utilities a common redirection. Of course, they could both be entered on their own input line with a specification of the required redirection (remembering to add the append symbol [+] to the second input line), but the action is easier entered as:

```
(dir; procs) >output.file
```

4.1.4 Command Piping

Piping is the ability to pass the output of one command as the input of another and may be performed by separating each of the commands with an exclamation point (!). For example, suppose you are interested in knowing how many shells are currently being run on your system. The procs command supports the flag -e, which allows the utility to show you all processes currently running; the grep utility acts as a filter, searching its input for some keyword, displaying only the input lines containing that keyword; and the count utility counts the number of lines of standard input received and outputs that number. The trick is to allow these commands to run simultaneously, allowing each to get information from another. Examine what has been recorded in Figure 14.

For more information concerning pipes, see Chapter 8, Section 8.1.

FIGURE 14 Command Piping

```
$ procs -e ! grep shell
  6   73   1.12     128   28.00k   0 w   0.40  0:10 shell <>>>t30
 14   74   1.138    128   28.00k   0 s   0.47  0:23 shell <>>>t63
 20   73   1.36     128   28.00k   0 s   0.41  0:22 shell <>>>t31
 25   22   1.12     128   28.00k   0 w   0.02  0:10 shell <>>>t30
 26   73   1.51     128   28.00k   0 w   0.34  0:21 shell <>>>t10
 66   74   1.140    128   28.00k   0 s   0.41  0:21 shell <>>>t50
 76   74   1.75     128   28.00k   0 w   0.74  1:16 shell <>>>t43
 78   63   0.142    128   28.00k   0 w   1.89  0:35 shell <>>>term
 84   73   1.16     128   12.00k   0 w   0.29  0:30 shell <>>>t12
 96   76   1.59     128   12.00k   0 w   0.67  0:36 shell <>>>t43
102   95   1.16     128   12.00k   0 w   0.09  0:29 shell <>>>t12
120   96   1.59     128   12.00k   0 w   0.18  0:36 shell <h0 >>>t43
$ procs -e ! grep shell ! count
"standard input" contains 13 lines
$
```

4.1.5 Process Priority

OS-9 gives you the ability to assign priorities to each process on the system. The priority for a process determines the amount of CPU time it gets before the system allows another process to run. (Since OS-9 is a time sharing system, each process runs for a short period and is then time-sliced, meaning another process gets to run for a short while.) Priority is a number between 0 and 65535 inclusive, where the higher the number is above the priority of other processes on the system, the more CPU time will be awarded. For detailed information concerning process scheduling, see Chapter 11.

When a process is started it is usually given the same priority as the parent process (which is most often the shell). The shell allows you to specify a different priority at execution time with the caret (^) operator, commonly found as shift-6 on most keyboards. To specify a priority simply add the caret, followed by the priority value on the command input line. For example, the command:

```
eatCPU ^20
```

specifies that the program eatCPU is to be run with a priority of 20.

The default priority on most systems is 128, meaning users shells are set up to start at a priority of 128. As long as all processes on the system share the same priority value, they will command an equal percentage of CPU time. What matters is not the priority value, but the relative difference from other process priorities on the system.

4.1.6 Stack Space

All computer programs require a stack (a First-In, Last-Out data area to hold temporary information) to continue to execute. On systems that support only one process the issue of stack size is a simple one: the stack starts somewhere high in memory and works downward, and other program data is placed low in memory. So as long as the system has enough memory to insure the two areas never meet, the program will run fine. More sophisticated systems, such as OS-9, which support many simultaneous applications, must be more careful about how stacks are placed in memory. To avoid memory corruption, each process must have its own stack in a unique area of memory. (Part of the time-slice operation is to change the CPU stack pointer from one process' stack to another.)

The job of maintaining a separate stack for each process is not a difficult one for the OS-9 system, but to do so the system must have a hint as to how much memory is required for each process' stack. To provide this information, each executable command has a maximum stack size built into it. If the process should exceed this space, problems will occur; either the system will notice that the stack has been overrun, and in this case terminate the offending process, or data will become corrupt. (Which problem depends on how the application was built, with or without stack checking enabled. The default is to include stack checking.)

Applications are built to include a default of 3Kbytes of stack, but can be built for more or less also. Additionally, there is an operator to override the stack space allocation for an application on a per-use basis. This operator is the pound sign (#) and is used to specify an additional number of kilobytes to add to the process, beyond what the utility was built with. The additional stack space is only applied to the new process, not the utility. Thus, it is possible to have two processes running the same utility, each with a different amount of stack space than the other.

As shown in Figure 15, the program recurse is run twice. The first time the system terminates it due to a stack overflow, while the second it runs until completion.

FIGURE 15 Giving Additional Stack Space to a Process

```
$ recurse
Starting Recursion
**** Stack Overflow ****
Error #000:002
$ recurse #110
Starting Recursion
Starting Routine 1
Starting Routine 2
Starting Routine 3
Starting Routine 4
Starting Routine 5
Starting Routine 6
Starting Routine 7
Starting Routine 8
Starting Routine 9
Starting Routine 10

Normal Termination
$
```

The number used after the pound sign is the number of kilobytes to add to the stack space. This number may also be optionally followed with the letter k, for readability purposes.

4.1.7 Wild Card Characters

For commands using file names as parameters, the shell has two wild card characters which simplify file name specification: the asterisk (*) and question mark (?). The asterisk is used to take the place of any number of characters (0 to 28 characters) and the question mark is used to take the place of exactly one character. Figure 16 shows the possible uses with five files: a, f, f1, f2 and file.

FIGURE 16 Wildcard Usage

```
$ dir
                    Directory of . 08:49:02
a          f        f1       f2       file
$ dir f*
f          f1       f2       file
$ dir f?
f1         f2
$ dir f?*
f1         f2       file
$ dir ?
a          f
$
```

Wild cards on a command line can present a nuisance under certain circumstances. If the character is not meant to be expanded as file name characters, but instead is to be sent to a utility as a command argument, the shell must be aware of this. The shell will attempt to expand any wild card character that is not enclosed within quotes. If you wish to change your prompt to an asterisk, you must enter the command as:

```
setenv PROMPT "* "
```

4.1.8 Command Aliasing

The standard shell from Microware does not have the ability to alias a command; however, Microware's MShell product does, through the assign command. For example, assign ls dir would allow you to type ls rather than dir to get a

directory listing. There are two problems associated with this, however: assignments do not change parameters accepted by a command; and although normal commands are case insensitive, assignments are sensitive to the input character case.

4.2 Built-in Commands

The OS-9 shell has a number of commands built into it. These commands are recognized by the shell before the system searches for the location of the command. We have already seen some of them: chd, chx, setenv and unsetenv. More of them follow:

4.2.1 ex

The ex command is followed on the command line by any legal command. The shell is then replaced by the command specified. For example, suppose you are running shell and would rather be running MShell (another command interface):

```
ex MShell
```

This command would terminate shell while starting MShell.

4.2.2 kill

The kill command takes one argument, a process id number. This process is sent the S_KILL signal (signal 0) and, if the process specified is a user state process owned by the same user as the shell, it is terminated. Kill may also be called with the parameter 0. In this case, all of the processes owned by the same user as this shell will be sent the S_KILL signal.

4.2.3 logout

This command terminates your shell.

4.2.4 profile

This command has a single parameter, the name of a text file. The specified text file is a list of commands that could be typed into the shell command line also. The shell will execute each command and, upon completion, get the next typed command from the shell's standard input. Any changes made to the environment within the file being profiled are also changed in the users environment.

4.2.5 setpr

The setpr command allows you to change the priority of a running process. It takes two parameters: 1) the process id of the process whose priority you wish to change and 2) the new priority. To change the priority of the process, you must either own the process or be a super user.

4.2.6 w

The command w forces your shell to wait until one of its background processes terminates. Once this condition is met, the shell will present the command input prompt. There is no method of waiting for a particular background process to terminate if there are more than one.

4.2.7 wait

This command forces the shell to wait until all of its background processes terminate before getting more input from the user.

CHAPTER 5

OS-9 Memory Modules

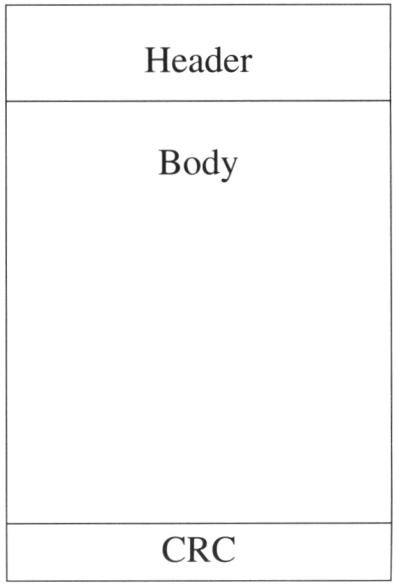
This chapter explains the composition of the OS-9 system.

5.1 OS-9 Building Blocks

The OS-9 system and its components are built on a modular level. When new items are added to the system, an OS-9 module is loaded into memory.

All modules have three sections: the header, the body and the CRC, as shown in Figure 17.

FIGURE 17 Module Format



5.1.1 Module Header

The header of a module is the first section containing information about the type of module listed, the size of the module, the name of the module and much more information which will be presented later in this chapter. The size of a module header is dependant upon the type of module, but is guaranteed to be at least 48 bytes. The first 48 bytes of any module header are described by the structure modhcom from the standard C include file module.h.

Modules currently in memory are located within a pseudo directory called the module directory. The system will give a listing of all modules in memory as response to the mdir command. The -e flag added to this command specifies that extended information is to be reported. Output provided with the -e option is: address, size, owner, permissions, type, revisions, edition number, link count, and edition.

FIGURE 18 Extended Module Directory Information

```
$ mdir -e
    Addr   Size    Owner Perm  Type  Revs   Ed #   Lnk Module Name
------------ ------ --------- ------ ------- -------- ------- ------ ----------------
00009800  28292     0.0  0555   Sys    a000     244    2    kernel
00010684   5706     0.0  0555   Sys    a000      25    1    ioman
00011cce    926     0.0  0555   Sys    8000      39    4    init
```

Note: Most of the output from this command has been omitted for brevity.

Most of the information returned from an mdir -e command is stored in the header of the module, but some of the information is dynamic. (The module address and link count are not permanent values.)

5.1.1.1 Name

The first field of information starting from the right, is the name of the module to which the rest of the line pertains. The name of a module is located within the module's body and the header only contains an offset into the module for where the name begins. This is because module names are of an arbitrary size and would not lead to a specific header size if located within the header.

5.1.1.2 Link Count

The next field to the left is the link count for each module. The link count gives an approximation to how many different things on the system are currently using this module. (The "things" that use modules depend on the type of module: processes use program modules; devices use file managers, device drivers, and device descriptors, etc.) Since memory is a valuable commodity and it is wasteful to keep a module in memory any longer than is necessary, modules are removed from memory once their link count reaches zero. The link count for a module is incremented each time its use is requested and decremented each time it is properly returned to the system. Note, that the link count is not guaranteed to be accurate, since there are several ways of artifi-

cially affecting it. For example, see the link and unlink commands at the end of this chapter.

5.1.1.3 Edition

The edition value of a module is a static value located in the module's header. This number is a 16 bit unsigned value which is generally assigned to the module at the time of creation. The system does not care what the edition of a module is at any step; rather, this value is used to aid users in identifying specific versions of a module. When Microware releases a new version of a product, any modules which have changed are sure to have increased edition numbers. When a Microware customer calls their technical support number, the support engineer will ask for the edition of all pertinent modules, possibly identifying out-of-date software.

5.1.1.4 Attribute and Revisions

This field is called Revs in the mdir output, but is actually comprised of two fields. Attribute and Revision are each 8 bit fields combined into one 16 bit quantity, with attribute occupying the most significant 8 bits. Attributes are defined by the bits turned on in that field. At the time of this writing only the three most significant bits have any meaning. The possible values are shown in Table 2.

TABLE 2 Attribute Values and their Meanings

Hex Value	Meaning
0x80	ReEntrant
0x40	Sticky
0x20	System State

Hex value is the hex code for the upper 8 bits of the Attribute/Revision field

The reentrance attribute is common to a major percentage of modules found in the OS-9 system. This operating system makes it fundamental that programs can be used by

multiple processes simultaneously, while not requiring programs to modify their code while running. A reentrant module is one containing code that does not change on the fly. Also, reentrance requires that the module not have any changeable variables embedded within the code rather than the data section. (It is easy to see why a program which modifies itself should not be considered to be reentrant, since if the program changes on behalf of one process it will have a potentially dangerous effect on another.) Modules containing code which are generated by Microware's high level language compilers will usually be reentrant. Code written in assembly language should be reentrant but that responsibility is left to the software engineer.

The attribute of "stickiness," also called the "ghost" property, determines if a module will be released from memory when its link count reaches zero. Certain modules are expected to be used rather often, for example, the dir module of the dir utility. If this module were to be loaded into memory for each use, a good deal of unnecessary time would be spent finding and reading this program. Instead, this and many other utilities from Microware have the attribute of being sticky, which causes the module to remain in memory once the link count reaches zero. The module will be removed from memory when it is implicitly unlinked to a count of negative one, or when the system needs the memory it is occupying for another purpose.

The state attribute can either be system (the bit is turned on) or user. Most utility-type programs are user state modules. System state, corresponding to supervisory state in the 68k CPU, is generally reserved for items such as device drivers and other operating system extension modules.

The revision byte does not have specific meaning. However, the revision value is significant and comes into play when a module is loaded from disk. The load command is used to implicitly place a module (stored in a file on disk) into memory. When this command is executed, the module may already be in memory. OS-9 does not simply ignore the load

request when this happens, but looks at the module stored on disk, specifically looking at the revision in that module. If the Revs field has a greater unsigned value than that of the module already in memory, the module on disk is placed into memory and the system releases knowledge of having the old module. (However, any programs currently using the old module will still use that module. It is only new requests for access to the module which will be diverted to the new one.) If the revision on disk is equal to (or smaller than) that of the module in memory, the link count of the module in memory is incremented and no further action takes place.

The attribute/revision value is placed within the module's header at the time of creation but may be changed at a later point.

5.1.1.5 Type

The type field is another combined field. This value holds the type of module in the upper 8 bits and the module language in the lower 8 bits. Valid module types are found in Table 3. The language field is mainly unutilized, with the language usually found either as object code or data.

TABLE 3 Module Types and their Definitions

Type	Definition
Program	The standard application created by the compiler.
Data	OS-9's method of shareable memory. Also, a generic module.
File Manager	A high-level driver for a certain class of device, such as "sequential character."
Device Driver	A driver for a specific device; usually needs a File Manager to operate.
Device Descriptor	Data pertaining to a particular device; refers to driver and manager.
Trap Handler	One form of a shared code library; relies on trap exceptions to work.
Subroutine Module	Another type of shared code library; no hardware exceptions needed.
System Module	The OS-9 Kernel and its extensions.

5.1.1.6 Perm

This field is a hexadecimal representation of the permissions associated with the module. There are three groups of read, write and execute permissions within this value, each group taking up one hex digit. The groups, from right to left are:

- owner — the owner of the module (see Section 5.1.1.7)
- group — users in the same group as the module owner
- world — users not in the same group as the module owner

Within each group, permission bits from least to most significant are read, write and execute. The fourth bit in each group is not used.

5.1.1.7 Owner

This field contains the group number and user number of the user who owns this module.

5.1.1.8 Size

This field tells the size, in bytes, for this module.

5.1.1.9 Address

This field tells the memory location where the module begins. Address is filled in at the time the module is loaded into memory and may or may not be the same with two successive loads. Since OS-9 code modules are typically built to be position independent, this is not a problem.

5.1.2 Module Body

The body of a module is the reason it has been created. If the module is, for example, a program module, the body contains the actual program. If the module is a data module, the body contains the actual data.

5.1.3 Module CRC

The CRC, or cyclic redundancy checksum, is a measure the system takes to insure the integrity of data within a module. Before a module may be entered into the module directory, it must pass a CRC check. That is, the computed CRC must be equal to the CRC stored in the module. If this is not the case, OS-9 will refuse to recognize the corrupted module.

5.2 Other Standard Module Oriented Utilities

5.2.1 load

The load utility is used to place the modules from a file into memory. The required parameter to the load utility is a file

containing one or more modules. This file name is, by default, relative to the current execution directory or, if not found there, to the directories specified by the PATH environment variable. If the file to be loaded is located in the current data directory, the -d flag must be used.

5.2.2 save

This command allows one or more modules in memory to be saved into one or more files. Parameters needed are the names of modules to be saved. By default, one file will be created for each module called out, with each file name the same as the name of the module. If only one file is needed to contain all modules specified, the `-f` flag can be used. This flag needs a parameter of its own: the name of the file to create to store the modules in. The command is specified as:

```
save module1 module2 module3 -f=filename
```

5.2.3 ident

This command is used to gather identifying information about the modules in a file. The information reported comes mainly from the header of each module. Ident can also be used to gather the same information for modules in memory with use of the -m flag. Ident differs in the output from mdir -e in that the information as specific to the type of module and value meanings (such as the various attribute bits) are spelled out in plain English.

5.2.4 fixmod

Fixmod is used to change some of the information in the header of a module including the edition, attribute/revisions, permissions and owner. It also allows you to update the CRC of the module and, in later versions of the command, the name of the module.

5.2.5 moded

This command is a highly specialized file editor. It has been written to allow easy editing of certain types of modules as described in the file /dd/SYS/moded.fields. Most device descriptors have field information in this file as does the init module. The information that is changeable is listed next to the command by field names to allow for an informed editing session.

➤ This utility does not currently exist in the OS-9000 operating system.

5.2.6 link

The link utility is used to artificially increase the link count of the specified module by one. This is something you may want to do to insure that the module will remain in memory, since the higher the link count, the more difficult it is to remove the module. (Recall, the module will at lease remain in memory until the link count has reached zero.)

5.2.7 unlink

This is merely the inverse of link; it causes the link count of the specified module to decrease by one. When the module's link count reaches zero, it might be removed from memory.

One reason a module may remain in memory with a link count of zero is the ghost attribute bit. If this bit is turned on, the module is considered "sticky" and will remain in memory until the space is needed by the system for other purposes, or until the link count reaches an effective -1.

➤ The system's init module has a compatibility bit which disables the sticky bit for all modules.

Also, a module may remain in memory with a link count of zero when the system *knows* it is being used. For example, if I am using the dir utility (and thus the dir module), you may attempt to unlink. However, since the dir module is the

primary module for my process (and the system can find this information without looking at the module link count), the module will remain in memory. Once my process (and all others using the dir module) completes, if the link count is still zero, the module will leave memory.

Occasionally, a file will contain more than one module. If the contents of a file are loaded into memory, the first module in the file is given a link count of one while the remaining modules have initial link counts of zero. The system records that all modules in the file have been loaded as a group. Since these modules were loaded as a group, they may only leave memory as a group. Thus, if a module has a link count of zero but other modules in its group have non-zero link counts, all modules will remain in memory.

Finally, if a module was found in ROM during the startup procedure, it will not be unlinked from memory. Of course, to get rid if it would be fruitless since the ROM could not be reclaimed for other system purposes.

CHAPTER 6

OS-9 Devices

This chapter will teach you how OS-9 organizes the controlling of devices.

6.1 The Components of OS-9

Figure 19 shows a high level view of OS-9. The system is shown to be very modular because that's exactly what any OS-9 system is, no matter how customized.

6.1.1 The Kernel

At the heart of the system is the system kernel. All other system components are attached in some way to the kernel. Thus, the system kernel is an overseer of all system operations.

6.1.2 User Applications and Utilities

At the top of Figure 19 we see the User Applications and Utilities box, representing all standard utilities on the system as well as any utilities that are non-standard.

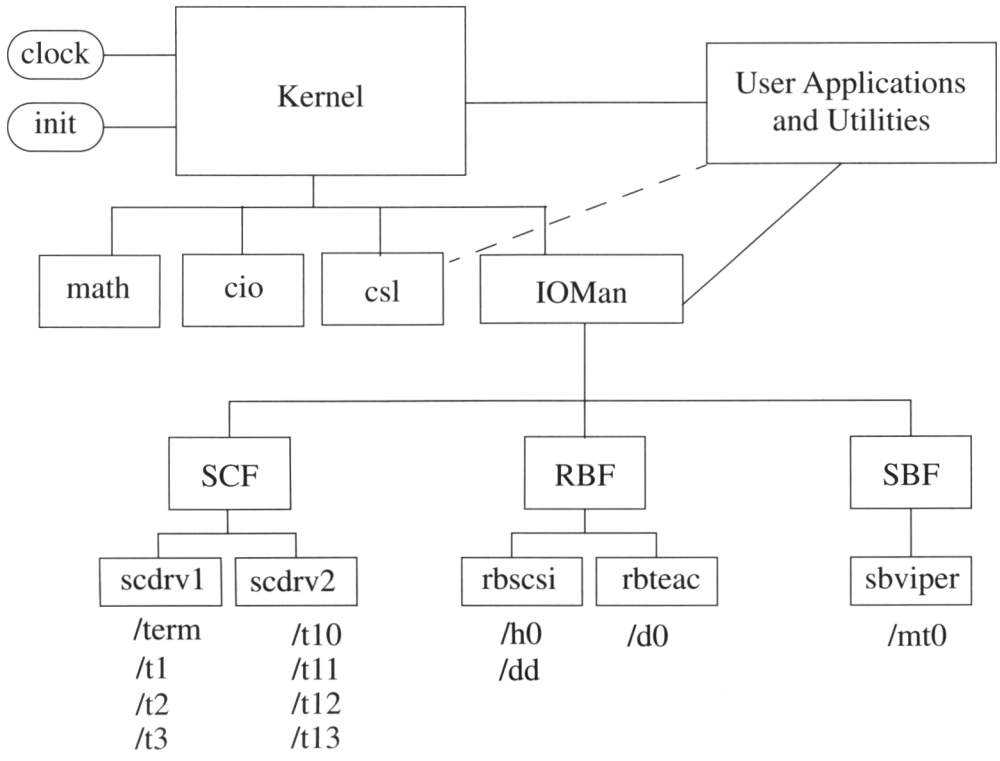
6.1.3 init and clock Modules

To the left of the kernel we see the system init module and clock device driver. The init module is needed upon starting up the system, as the kernel consults this module for information pertaining to how the system will run. The clock device driver is needed to configure the system's timer interrupts. This driver may be a dummy driver if no interrupting clock hardware is available, but this is rarely the case. The clock is one of the few drivers that does not fit the standard model of a device driver, as will be explained later.

6.1.4 Trap Modules

Below the kernel, we see the csl trap module. A trap module contains code and is generally thought of as a shared library. csl, the C subroutine module, is made available to programs compiled under Ultra C with the -i flag. This trap module allows applications that perform I/O (all applications?) to contain less space, at the cost of slightly more overhead to performing I/O operations.

FIGURE 19 A High Level View of OS-9



6.1.4.1 cio

Figure 19 shows the three standard trap modules available on the OS-9 system. cio is the C input/output trap handler used by the previous C compiler, version 3.2, whenever a program is compiled with the -i flag. This module handles functions like printf() and scanf(); that is, the code for these and other I/O functions resides in the trap module. The application code only contains calling mechanisms to the trap module. (For more information on trap modules, see Chapter 13.)

6.1.4.2 csl

The csl trap module is a replacement for cio and is used by applications compiled with the -i flag by the Ultra C compiler. If you are using programs compiled with both Ultra C and the version 3.2 compiler on the same machine it may be necessary to have both the cio and csl modules in memory.

6.1.4.3 math

This trap module contains code for performing standard floating-point co-processor functions. The standard OS-9 system comes with two math trap modules, one in a file called math and the other in a file called math881. The math trap module is included with programs that are compiled with the -x flag by the 3.2 compiler and is not needed by Ultra C.

➤ The logic behind two math modules: code that was compiled with the -x flag was written so all floating-point math functions would go through the math module. This would allow one program to run on all systems whether or not an FPU was present. Systems that had an FPU would place the module from file math881 in memory and this module would simply call the FPU while systems without an FPU would place the module from the file called math into memory. This module would emulate the co-processor call.

6.2 Devices and their Drivers

Finally, below the kernel we see the elements of the I/O system. Every (standard) device on an OS-9 system contains three items: a file manager, a device driver, and a device descriptor.

6.2.1 File Managers

File managers are basically generic device drivers which handle all devices of a certain class or family. Examples of device families include serial character devices (for standard terminal ports), random block devices (disk drives) and sequential block devices (such as tape drives). A file

manager does not know anything about the hardware a system has, but it does know what the kernel expects in I/O to and from a device. Since most of the standard code for a device is resident inside a file manager, the software engineer is relieved of the burden of placing it inside the device driver. Thus, a file manager is the intermediary between the kernel (who expects things to be in a certain way) and device drivers (which understand a particular device).

6.2.2 Device Drivers

Device drivers are written to accommodate: a) a specific file manager (since all file managers may have different variable and parameter conventions) and; b) a specific piece of hardware. Drivers are written to conform to the specifications of the *Microware I/O Technical Manual* to allow them to interface with a particular file manager. As seen in Figure 19, a device driver may handle more than one device at a time; the SCF device driver shown is currently handling four. OS-9 device drivers are written to be reentrant to accommodate this. Each physical device has its own static storage area so the driver does not become confused about which device it is handling at any point in time.

6.2.3 Device Descriptors

Device descriptors are the final element in OS-9 devices. They are modules which contain information rather than code. (Some of the information within them includes the names of their associated device driver and file manager.) The information within them is laid out in a manner specific to the file manager controlling them so both the file manager and the driver can find what is needed. Some of the information they contain includes the hardware's physical address (so the driver can find it), the interrupt vector the hardware should generate (so the driver can initialize it properly) and the interrupt priority. (See Chapter 15 for information on interrupts and interrupt priority.)

6.3 Accessing Devices

To access any device you must know its name which is identified by a slash followed by the name of the device descriptor. For example, to send the output of a dir command to the device represented by the "t2" descriptor, output would be redirected as:

```
dir >/t2
```

If the slash were not present, the system would try to create a file called t2 in the current data directory.

Before a device can be used, it should be initialized; however, if a device is not initialized prior to use, it will be initialized as soon as a link is made to it (provided all elements to the device are in memory first). Allowing devices to be initialized at the time of their first use has one serious drawback: the device is uninitialized when the last link to it goes away. This functionality can make some devices seem to operate much more slowly than they should!

The command devs lists all initialized devices on the system. If a device you wish to use is not initialized and you would like to explicitly initialize it prior to use, the command iniz followed by the device descriptor name will initialize it (provided that the file manager and device driver can be found in memory). If a device will no longer be used and you wish to implicitly un-initialize it, the command deiniz will accomplish this.

CHAPTER 7

Paths and Processes

This chapter introduces you to usage and manipulation of device I/O channels — paths.

A path is the channel used to perform any input/output on the system. All paths are represented by a number between 0 and 31 inclusive to user state processes, thus a user state process may only maintain 32 open paths at any point in time. (System state processes are limited only by system memory in the number of channels they may use at a time.)

7.1 Path Descriptors

The number of each path represents a 256 byte system data structure called a path descriptor which OS-9 uses to hold information pertaining to the particular path. This information includes the actual device descriptor/driver/file manager related to the path, the number of processes using the path, access modes the path was opened with and the memory location of static storage area for the particular path's driver.

7.2 Standard Paths

When most processes are started they begin with three open paths: the standard input, standard output and standard error output paths. These paths take positions 0, 1 and 2, respectively. No matter what path is in position zero, that path is considered to be standard input.

This being the case, the next path used by a process will always be in position three. This is because every new I/O channel added to a process will be positioned in the lowest numbered slot that is currently unused. Thus, if you wish to change the standard output of your own process after it begins running, the program can be written to close path 1 and then open a path to the device you wish the standard output to point to.

7.3 Basic C Calls for Paths

7.3.1 error_code _os_open(char *name, u_int32 mode, path_id *path)

The open call is used to open a path to an existing file or device as specified by the first parameter to the call, a character pointer to the pathlist (name of the device). The second call parameter specifies modes the path is being opened for, reading and/or writing. If a path is opened for reading and the write function is called specifying the path, the write request will fail. The final parameter to the call is a pointer to where the path id should be placed. This path id must be used for all other commands referencing the path.

Modes used for opening a path are symbolically defined in the standard header file modes.h. Some of them include S_IREAD and S_IWRITE, to specify read and write access, respectively. The mode S_IEXEC may be specified to indicate that the file is referenced from the execution directory.

7.3.2 error_code _os_create(char *name, u_int32 mode, path_id *path, u_int32 perm, ...)

This call is used to create a new file. When a file is created, it is opened at the same time. For devices which support multiple files, the file is created within the device's directory structure. For devices which do not support multiple files, such as unnamed pipes, (see Section 8.1.1) create is synonymous with open.

The first three parameters to the create call are identical to those of the open call. There is a fourth mandatory parameter which specifies the file permissions for the file being created. These permissions come from the same include file as the access mode choices.

In specifying modes for this call, there are three special modes which do not affect how the file will be accessed, rather how the file will be created.

7.3.2.1 S_IEXEC

This mode specifies that the file being created is to live in (or be relative to) the current execution directory.

7.3.2.2 S_EXCL

If this mode is included, an error will be returned from the create call when the file specified already exists. If the mode is missing and the file being created already exists, the file will be emptied.

7.3.2.3 S_ISIZE

This option in the modes parameter allows the programmer to pre-allocate space for the file being created. If a program is going to log a megabyte of data to a disk file, the program will run much quicker if all of the space for the data is allocated at the time of file creation. If space is not allocated, the system will constantly be adding small amounts of space to the file. Pre-allocating file sizes also helps to keep disk fragmentation to a minimum.

One problem with creating and pre-sizing a file is that if there is not enough free space on the disk in *one* memory fragment to satisfy the creation request, the file will be pre-sized as large as the largest fragment found. However, there is a subroutine, _os_ss_size(), whose parameters include a valid open path number and a size integer, which sets the size of the file to the size specified. The data in the file (beyond any data written into the file) is undefined data that happens to be on the disk. This call may also be used to truncate a file to a shorter size.

7.3.3 error_code _os_dup(path_id orig, path_id *copy)

This call allows a program to duplicate one of its open paths. The "orig" path will be duplicated to the "copy". It may not be immediately obvious what use a duplicated path provides, but a discussion will follow in Section 7.4. The parameters to this call consist of the path id to duplicate and a pointer to another path id, which will hold the id of the copy of the first path. Just as in opening and creating paths, the duplicate path will occupy the lowest numbered unused path.

7.3.4 error_code _os_close(path_id)

When a process no longer needs a certain path, that path should be closed. This call takes a path id and closes it. If a user state process terminates without closing paths, they will be closed by the system.

7.3.5 error_code _os_read(path_id path, void *buffer, u_int32 *count), _os_readln()

These calls each provide for reading data from a path that has been previously opened. Each call takes identical parameters: a valid path id, a buffer location to place the incoming data and a pointer to an integer representing a byte count.

The read call will read the number of bytes specified by the count variable into the buffer. If there is an error reading all of the data or a deadly I/O signal is sent to the process (see Table 4), the count variable will contain the actual number of bytes that were read.

The read-line call works similarly to read, however, the count variable is only a maximum number of bytes to read. The call will stop reading data when an end-of-line character is seen. This call also allows line editing, such as backspacing, to occur.

7.3.6 error_code _os_write(path_id path, void *buffer, u_int32 *count), _os_writeln()

The write and write-line calls are the counter-parts to read and read-line. The only difference is data is now going to the device rather than coming from the device.

7.4 Path Manipulation

When a process spawns a child process, that child generally gets a duplicate of the first three paths of the parent. If the parent would like to give the child different standard paths than its own, it must first modify its own standard paths to resemble the path structure desired for the child, then spawn the child process. After the child is running, the parent may restore its own standard paths. The duplicate path call allows the parent's final step of restoring its standard paths.

Listing 3 contains a simple program that will spawn a child with different standard output than the parent.

Listing 3. Spawning a Child with Different Standard Output from the Parent

```
/*******
** newoutput.c
**
** This code will not compile because a
** fictitious subroutine has been
** included: spawn(). The intention is
** that if this routine did exist, it
```

```
** would create our child process.
**
*******/
#include <stdio.h>
#include <modes.h>
#include <types.h>
#include <errno.h>

main()
{
   path_id  new_output_path, save_stdout, dummy;

   /* the printf() call always sends its */
   /* output to the standard output path */
   printf("This is my normal standard output\n");

   /* first, create the path to a file */
   /* called "new_output_path" */
   errno = _os_create("new_output_path",
      S_IREAD|S_IWRITE, &new_output_path,
      S_IREAD|S_IWRITE);

   if (errno != 0) {
      /* An error occurred while creating */
      /* our path */
      fprintf(stderr,"Hmmm. Error %d occurred while
          creating our new path!\n",errno);
      _os_exit(errno);
   }

   /* duplicate stdout so I can restore it later */
   /* path 1 is always stdout, so ... */
   errno = _os_dup(1,&save_stdout);
   if (errno != 0) {
      fprintf(stderr,"Hmmm. Error %d occurred while
                duplicating stdout!\n",errno);
      _os_exit(errno);
   }

   /********
   now, make my first three paths look the way I want
   the child to look.

   My current setup is sure to be:
      0) my stdin
      1) my stdout
      2) my stderr
      3) the stdout I want the child to have
      4) copy of my stdout
   ********/
   _os_close(1);

   /* I know it will take slot 1 here! */
   _os_dup(new_output_path, &dummy);
```

```
    /* The last two calls just changed slot 1 to be */
    /* the stdout I want the child to have, so it */
    /* is now safe to spawn the child */

    /* remember, this is a fake call! */
    spawn(child_process);

    /* ok, now to restore my own standard paths */
    _os_close(1);
    _os_dup(save_stdout, &dummy);

    /* now, I might as well close my excess paths */
    _os_close(new_output_path);
    _os_close(save_stdout);

    printf("All done, exiting!\n");
    _os_exit(0);
}
```

7.5 Processes

As seen earlier, the procs utility is a good tool for finding information about all of the process running on the system at any point in time. The information this utility provides comes from a data structure associated with each process, a process descriptor. Every process on the system has its own process descriptor to store information such as the id number of the process, the id of the process' parent (if there is one), the id of the process' last child, path information, memory information, and lots more. (For full details, see the standard include file <procid.h> which defines a structure called procid.)

➤ Under OS-9000, the process descriptor contains space for multiple child id numbers, but under OS-9 there is only storage for the last child created. The children of a parent have their process descriptors linked together using an id field called a sibling id. A process' sibling id is the child id the parent contained just prior to spawning the new child.

To spawn a child process, a potential parent must make a call to the _os_exec() function:

[error_code _os_exec(u_int32 (*func)(), u_int32 priority, u_int32 pathcnt, void *modname, char **argv, char **envp, u_int32 stacksize, ...]

The first parameter to this function is the name of the function that will actually create the new process, and may be one of _os_fork(), _os_dfork() or _os_chain(). Which function is specified will depend on what should be happening. _os_fork() allows the parent to create a normal child which will run parallel to the parent. _os_dfork() creates a child that is under tight control of the parent and is reserved for programs which function as debuggers. _os_chain() allows the creating process to be terminated for the sake of having the child take over its process descriptor.

Other parameters to _os_exec() include priority, path count, module name, argument vector, environment pointer, stack size addition and others depending on which function is specified as the first parameter.

7.5.1 The func Parameter

The first parameter to _os_exec() is used to name the function that will do the work of creating the child process descriptor, or modifying the process descriptor in the case of _os_chain().

7.5.1.1 _os_fork

This function is the one normally used to create a new process. If used, _os_exec() requires three additional parameters: a pointer to the location to store the process id number of the new process and the Type/Language of the new process. The third parameter must be set to zero on OS-9.

➤ Under OS-9000, the third parameter is an integer that is either 0 or 1. 0 indicates that the new process is a child of the current process while 1 makes the new process an orphan, that is, having no parent process. This may be desired because if a process with a parent terminates, the process descriptor will remain on the system until either the parent is notified (through the _os_wait() call) or the parent terminates.

7.5.1.2 _os_chain

This function is specified when the calling process has no intention of continuing to execute after the process creation. Rather than create a new process descriptor the system will simple recycle the current process descriptor and the new process will take the place of the old one. This call requires one additional parameter to _os_exec(): the Type/Language of the new process.

7.5.1.3 _os9_dfork

This function is only used by controlling parents, generally debugger utilities. The first two additional parameters are the same as those used by _os_fork() plus one more: a pointer to storage for the register stack of the child.

Under OS-9, when a process is being debugged (as will be the case when _os9_dfork() is used), only one of the two processes involved (parent and child) may be active at any point. The system keeps the other process in a suspended state.

➤ OS-9000 uses the function _os_dfork() in place of _os9_dfork(), and requires yet one more parameter: a pointer to storage for the floating-point registers for the child.

7.5.2 Priority

This value is the priority at which the child should run. If this value is zero, the child will start running at the same priority as the parent. Thus, it is impossible to start a process at priority 0. (The priority of a running process may be changed with the _os_setpr() subroutine, however.)

7.5.3 Path Count

Until now it has been assumed that a child process inherits its parents standard paths. This parameter determines how many open paths are passed along to the child. It may be zero, three (to just pass the standard paths), 32 or any number in between. Whatever number of paths are chosen to be

passed on, the paths given to the child are always the lowest path count paths.

7.5.4 Module Name

This parameter is simply a pointer to the name of the module that must be in memory for the process to execute. This is generally a pointer to the argv[0] element.

➤ OS-9000 has additional functions for the first parameter to _os_exec(). They are named similarly to those available for OS-9 but have an 'm' appended to the function name indicating that, rather than using a pointer to the module name for this parameter, the pointer will point to the actual module in memory. This saves the system from searching for the module. Of course, OS-9000 supports the three main functions as well.

7.5.5 Argument Vector

When a command is input with more than just the name of the utility, i.e., to request special options like extended information, that information must be seen by the utility. (Of course some options such as I/O redirection are parsed by the command interface [shell] and the application never sees them.) These options are passed to the process in an argument vector which is a parameter to the main() function. main() takes in two parameters. (There are three in Ultra C's extended mode: an environment variable pointer is the third. This value is also available as a global pointer described in Section 7.5.6.) The first parameter is an argument count integer often called "argc" and the second is an argument vector which is a double pointer to character information often called argv. For input of the form:

```
dir -e filename
```

the argc integer would be equal to 3. argv[0] would point to the text dir, argv[1] to -e, and argv[2] to filename, while argv[3] would point nowhere. The argv variable is what must be passed as the argument vector to _os_exec(); other system components will decide the value of argc.

7.5.6 Environment Pointer

As mentioned earlier, child processes inherit a copy of their parent's environment. This is done properly only if the environ parameter points to the parent's environment. Under Microware's Ultra C compiler, all user state programs may declare an external character double pointer called _environ; this pointer will point to the process environment variables. Example:

```
extern char ** _environ;
```

➤ Note, that beginning with Ultra C Version 1.2, a program should not explicitly declare any of the "standard" global variables provided by the compiler. Instead, they should include the <cglob.h> header file.

7.5.7 Stack Override

If the parent feels the child needs more stack space than has been compiled into the application, it will fill this value with the number of bytes to give the child additional stack space. This is how the shell handles the pound sign operator.

7.6 error_code _os_wait(process_id *pid, status_code *stat)

The wait call allows a processes to stop running until one of its child process has terminated. The parameters required by this call are two pointers: one for the location to place the id number of the process that terminates, and the other for storing the exit code of the process. There is no way to wait for a particular process.

The _os_wait() call normally does not return until a child terminates. However, if the parent receives a signal while _os_wait()-ing, the call will return. This event is indicated by a zero being placed at the location pointed to by the first parameter, the id pointer.

Listing 4. fork.c

```
/* fork.c */
#include <stdio.h>
#include <process.h>
#include <module.h>
#include <errno.h>

extern char ** _environ;


/* need NULL termination */
char *CmdArgs[] = {"dir","-e",0};

main()
{
   process_id id, cid;
   u_int32 status;
   u_int32 priority = 0; /* same as parent */
   u_int32 path_cnt = 3; /* standard paths */
   u_int32 extrastk = 0; /* no extra stack */
   u_int32 typelang = (MT_PROGRAM<<8)|ML_OBJECT;

   printf("forking child...\n");
   errno = _os_exec(_os_fork,priority,path_cnt,
      CmdArgs[0],CmdArgs,_environ,extrastk,
      &cid,typelang,0);

   if (errno != 0){
      fprintf(stderr,"Error forking child\n");
      _os_exit(errno);
   }

   printf("The child's id is %d\n",cid);
   _os_wait(&id,&status);
   printf("Process %d exited with status = %d\n",
      id,status);
}
```

CHAPTER 8

Interprocess Communication — Data Transfer Methods

Pipes and data modules — the tools OS-9 provides for passing data between processes — are learned from reading this chapter.

8.1 Pipes

A pipe is a special device which has no hardware of its own, rather it is a collection of memory. More specifically, a pipe is an area of memory which acts like a first-in, first-out queue. Typically, a pipe has one producer (data writer) and one consumer (data reader) though pipes may have any number of producers and consumers. A pipe is basically a serial-like device whose data is destroyed as soon as it is removed (read) from the pipe.

8.1.1 Pipe Types

OS-9 supports two types of pipes: named and unnamed.

8.1.1.1 Unnamed Pipes

An unnamed pipe may only be accessed by the process that creates it, or by processes that have the pipe as an open path

when they are created. The pipe created when one process is piped into another (see Section 4.1.4 on page 32) is an unnamed pipe. Since there must be some type of name for all devices opened or created on the system, the name "/pipe" is used to create a new pipe. Even though the name used for unnamed pipes is identical in all cases, a new pipe is received with all open or create calls to this device.

8.1.1.2 Named Pipes

A named pipe is treated in some respects like a file. One may be accessed by almost any process knowing the name of the pipe. Part of the likeness to files includes having file permissions, so a process may only access a named pipe if the permissions on the pipe allow it. Named pipes are also file-like in that they are located in a (pseudo) directory on the system, the pipe directory. The command:

```
dir /pipe
```

will return a directory listing of all named pipes currently on your system. The device name is "pipe" as evidenced by the slash preceding it. One major difference between this directory and a standard disk device is, while a disk may be multi-layered (have sub-directories), the pipe device is a flat file structure. Figure 20 shows an extended directory listing of the pipe directory.

FIGURE 20 The Pipe Directory

```
$ dir -e /pipe

                    Directory of /pipe 09:49:29
 Owner    Last modified  Attributes Sector Bytecount Name
-------  -------------  ---------- ------ --------- ----
$ echo "test data" >/pipe/pipename
$ dir -e /pipe

                    Directory of /pipe 09:49:46
 Owner    Last modified  Attributes Sector Bytecount Name
-------  -------------  ---------- ------ --------- ----
0.0     94/11/08 0949   ------wr  EAAA00        10 pipename
$ list /pipe/pipename
test data
$ dir -e /pipe

                    Directory of /pipe 09:50:20
 Owner    Last modified  Attributes Sector Bytecount Name
-------  -------------  ---------- ------ --------- ----
$
```

Notice that in this example there are initially no named pipes on the system. One is created by simply redirecting some output to a pipe specified by name and it shows up as having data in it. The pipe is then read from (by the list utility) and, once emptied, the pipe is removed from the system, as evidenced by the next extended directory listing of /pipe.

8.1.2 C Subroutine Calls for Pipes

The subroutines for opening, creating and path-manipulating pipes are the same as those discussed in Chapter 7. OS-9 has several calls that have been defined into Microware's implementation of the C language. These calls also work with paths other than pipes, though they may work somewhat differently. (These cases are noted below.)

8.1.2.1 error_code _os_gs_ready(path_id path, u_int32 *incount)

This call, taking a valid path number and pointer to 32 bits of storage as parameters, will return the number of bytes available on the path specified to the location pointed to by *incount. _os_gs_ready() may be used on SCF based devices in the same manner.

8.1.2.2 error_code _os_gs_eof(path_id path)

This call is used to determine whether or not a read on the selected pipe will cause an End-Of-File error for some process. If so, this call returns the E_EOF error value. The conditions under which a read of a pipe will result in E_EOF for a process are listed later in this chapter.

8.1.2.3 error_code _os_ss_sendsig(path_id path, signal_code signal)

The _os_gs_ready() call provides a mechanism for polling a device, but under many cases a program does not have the time to waste polling. To avoid this the _os_ss_sendsig() routine installs a pending signal on the pipe (of SCF device) specified by its path number. (Handling signals is discussed in detail in Chapter 9.) The installed signal will be sent to the process which executed the call as soon as data is available on the path for reading. (If there is data ready at the time of the call, the signal is sent immediately.) Once the signal has been sent, it would require a second request for a second signal, and there is no way to get a signal on "new" data available unless the "old" data is first removed from the device buffer.

If one process attempts to install a pending signal on a device while another process either has an outstanding pending signal on the same device or is blocked trying to read that device, the signal request will fail with an error indicating the device is busy. In the same light, if a process attempts to read a path which has a signal pending on it, the reading process will receive an error.

8.1.2.4 error_code _os_ss_relea(path_id path)

In some circumstances a process with a pending signal may no longer be interested in receiving that signal, such as when a timeout condition has been reached. The _os_ss_relea() call removes a pending signal from the path specified, as long as the _os_ss_relea() process is called by the same process which called _os_ss_sendsig().

8.1.3 Implementing a Timeout for Input with the Special Pipe (Path) Calls

Listing 5 contains a subroutine which will call the _os_read() function if data becomes available within the specified amount of time. This subroutine will not work under all conditions, so a more complete one is presented in Chapter 9.

➤ This routine also relies on having signal handling within your application, though what the signal intercept handler does when called is insignificant to our current purposes. More information concerning signals is found in Chapter 9.

Listing 5. timeout1.c

```
/* timeout1.c */
/************************************************
* The following program segment implements a simple
* timeout mechanism for receiving input from a pipe
* or serial device within a pre-defined amount of
* time. This first version of code does not work in
* all cases!
************************************************/

/*********
* Input parameters:
*   pn: the path number to get data from
*   buffer: the location to place data
*   count: the number of bytes to read
*   timeout: the number of seconds to wait for data
* Output values:
*   0: timeout was reached before input arrived
*   #: the number of bytes read in by _os_read()
*********/
int
readTimeout(int pn, int *buffer, int count,
   int timeout)
```

```
{
   _os_ss_sendsig(pn,1000);
   _os9_sleep(&timeout);
   if (timeout) {
      /* we've reached timeout */
      _os_ss_relea(pn);
      return(0);
   } else {
      _os_read(pn, buffer, &count);
      return(count);
   }
}
```

8.2 Potential Problems with Pipes

This section presents several common conditions that may be seen using pipes.

8.2.1 Writing to a Full Pipe

8.2.1.1 Named Pipes

An attempt to write to a named pipe that is full will block if there is no room to complete the I/O request. Once another process reads some of the data in the pipe, making space available for more data, the I/O request will continue. The writing process will not return from the write() call until either the request is complete or a signal is received by the process (while unmasked).

8.2.1.2 Unnamed Pipes

What happens when a write request is made to a full unnamed pipe depends on what other processes with access to the pipe are doing. If all processes with access to the pipe are trying to write into it (while it is full), this is a potential dead-lock condition. The only processes that may read data from the pipe (thereby freeing up space for more writes) are locked trying to write into the pipe, which will now never be emptied. OS-9 can sense this condition, however, and the

first process to have gotten in line for writing will receive an error from the write call.

8.2.2 Reading from an Empty Pipe

The effects of reading from a pipe that has no data available is identical between each type of pipe. If all processes with the pipe open are busy trying to read from it, the first reader in line will receive an error.

8.2.3 Creating a Pipe whose Name is Taken

If a creation request is made for a pipe with the same name as another pipe on the system, the request will fail just as trying to create a file that already exists will. Of course this cannot happen with an unnamed pipe.

8.2.4 Opening a Pipe that Does Not Exist

8.2.4.1 Named Pipes

An open request for a named pipe that does not exist will result in an error just as an attempt to open a non-existent file will.

8.2.4.2 Unnamed Pipes

Opening an unnamed pipe (rather than creating one) will create a new unnamed pipe on the system, 90 bytes in size. (See Section 8.3 for details on pipe sizes.)

8.2.5 Closing Non-Empty Pipes

8.2.5.1 Named Pipes

When a named pipe is closed and data remains, the pipe stays on the system and may be accessed by other applications at any time. If a named pipe is closed (by all processes having it open) and no data remains, the pipe is deallocated.

8.2.5.2 Unnamed Pipes

If multiple processes have an unnamed pipe open and one closes it, nothing significant happens. (Of course, the system notes the fact that one less process is using the pipe.) When the last process with an unnamed pipe open closes it, the pipe is deallocated and any remaining data is lost.

8.3 Pipe Sizes

When a pipe (either named or unnamed) is created, by default it is given enough memory to hold 90 bytes of data. This can be overridden at the time of creation if the S_ISIZE option is supplied to the _os_create() call.

8.4 Data Modules

Data modules are the preferred method of sharing memory across processes under OS-9. A data module is a module in memory whose body format is defined by the software engineer. Data modules allow programs to pass blocks of data to one another and these programs do not have to be hard-coded with an address of shared memory, rather they are coded with the name of the module, allowing the system to handle the details.

8.4.1 Shared Memory

Using a data module for shared memory purposes is quite trivial. The module must be created by the first process to come on-line, and subsequent processes must link to the module. When the first process creates the module, it specifies a size large enough to accommodate the data to be shared. The structure of this data is up to each process using the memory. (Of course, processes should be using the same format or a compatible format on the shared memory!)

8.4.2 Creating a Data Module

The function _os_mkmodule() takes as input the name, type, language, attributes, revision and permissions with which to create a memory module. Other input parameters include a pointer for the location to store the module pointer and another for the storage location of the data pointer (or execution offset for a non-data module.)

Although this call seems to imply that it could be used to create any type of module, according to the Ultra C Library manual section on this call,

> ➤ The created module always begins at offset $34 within the module. This implies that this call cannot conveniently make program modules, trap handlers, file managers, device drivers and device descriptors.

One final parameter to this call is specification of a certain area of memory to place the new module (assuming your system is set up for colored memory). For example, if your system has a special area of memory set aside for non-volatile RAM and you wish your data module to be active at all times, this memory color could be specified. Choosing memory area zero is a "don't care" specification.

The other call to create a module, _os_datmod(), is basically the same as _os_mkmodule() except the parameter for the selection of colored memory.

The following program segment will build a data module and report its address information. The data module created is large enough to hold an array of DATA_SIZE integers.

Listing 6. Creating a Data Module

```
/*** makemod.c ***/
#include <stdio.h>
#include <module.h>
#include <types.h>
#include <errno.h>

#define DATA_SIZE 100
#define DMOD_NAME "dat_mod"

main()
```

```
{

   u_int16 attr_rev, type_lang;
   u_int32 perm, dm_size, counter;
   mh_com *mod_head;
   int *data;

   /* declare space for DATA_SIZE integers */
   dm_size = DATA_SIZE * sizeof(*data);

   type_lang = (MT_DATA<<8);
   attr_rev = (MA_REENT<<8);

   /* only permissions are owner r/w access */
   perm = MP_OWNER_READ|MP_OWNER_WRITE;

   errno = _os_datmod(DMOD_NAME, dm_size, &attr_rev,
                  &type_lang, perm, (void**)&data,
                  (mh_data**)&mod_head);

   if (errno==0) {
      printf("The module %s is located at 0x%x\n",
               DMOD_NAME,mod_head);

      /* initalize data */
      for (counter=0; counter<DATA_SIZE; counter++)
         data[counter] = counter;
   } else {
      printf("Sorry, an error occured during module
               creation!\n");
      _os_exit(errno);
   }

   /* at this point, we can use the data module as often */
   /* and for as long as we wish. */

   /* poke 500 into the module... */
   *data = 500;

   /* we should unlink from the data module before */
   /* exiting, but we don't know how to yet! */
   _os_exit(0);
}
```

If this program fails in its attempt to create a data module, it will exit with the error code returned. Creating a module will fail if, for example, the data already exists or there is not enough memory for the module.

8.4.3 Linking to a Data Module

If a module the program needs to access is already in memory, it must only be linked to. This is accomplished via the _os_link() function. The parameters specified are similar to those used by the _os_datmod() function, except module size and permissions. Another difference is that the name parameter is a pointer to a pointer this time, due to compatibility with OS-9000. Also, there is no memory color specification. Listing 7 is a modified version of the program above, this time expecting the module to exist in memory.

attr_rev does not need to be initialized but is filled with the actual attribute/revision word contained in the module header upon return of _os_link().

type_lang may be set to zero, indicating a "don't care" state. In this case, type_lang will be filled with the actual type/language word from the module header.

Listing 7. Linking to a Module in Memory

```
/*** linkmod.c ***/
#include <stdio.h>
#include <module.h>
#include <types.h>
#include <errno.h>

#define DATA_SIZE 100
#define DMOD_NAME "dat_mod"

main()
{

   u_int16 attr_rev, type_lang;
   u_int32 counter;
   mh_com *mod_head;
   int *data;
   char *d_name;

   d_name = DMOD_NAME;
   type_lang = (MT_DATA<<8);

   errno = _os_link(&d_name, &mod_head,
      (void**)&data, &type_lang, &attr_rev);

   if (errno==0) {
      printf("The module %s is located at 0x%x\n",
```

```
                                        DMOD_NAME,mod_head);

        printf("The attribute/revision word is
                                              0x%x\n",attr_rev);

        /* initalize data */
        for (counter=0; counter<DATA_SIZE; counter++)
           data[counter] = counter;
     } else {
        printf("Sorry, an error occured during
                                    module linking!\n");
        _os_exit(errno);
     }


     /* we should unlink from the data module */
     /* before exiting, but we don't know */
     /* how to yet, so just comment it: */
     /* _os_unlink(mod_head); */
     _os_exit(0);
}
```

8.4.4 Loading a Data Module from Disk

When a module has been stored to disk, it may be loaded into memory and simultaneously linked to. The location of the file holding the module(s) to be loaded may be either the current data directory or the current execution directory / along the PATH environment variable, depending on which of the two calls is used (see below).

If the file specified contains more than one module, all modules are loaded into memory. In any case, a link is also performed to the first module in the file.

The call _os_load() takes the same parameters as _os_link() with two extra: a colored memory and a mode qualifier. The mode qualifier determines the base disk location for relative file names (those not beginning with a slash.) By default, _os_load() bases relative filenames from the execution directory, but the S_IREAD mode forces relative access from the data directory.

_os_loadp() loads the modules in a file found in either the current execution/data directory (determined by the mode)

or along one of the PATH directories. The parameters specified are the bare minimum: file name, mode qualifier, a pointer for a character string and the address to store the module pointer returned. The character pointer passed as the third parameter will be filled with the pathlist to the file loaded into memory.

8.4.5 Saving a Data Module to Disk

There is no included subroutine to save a memory module to disk, but it is a relatively easy thing to do. One method is to call the system() routine which takes as a parameter a character string containing any command that could also be typed as input to the shell prompt, such as save modulename.

The other method is one of brute-force: write a subroutine which creates a file, write the entire contents of the module (starting from the module pointer) to the file and close the file. See Listing 8.

Listing 8. Subroutine to Save a Module

```
/*** savemod.c ***/
/* This is not a program, just a subroutine! */
#include <modes.h>
#include <types.h>
#include <errno.h>
#include <module.h>

/****
** Input parameters:
**     mod_ptr: pointer to the module to save
**     fname: name of the file to create
**     perm: permissions to create the file with
**
** Returns the error code associated with file
** creation, if any.
****/
error_code ModSave(mh_com mod_ptr, char *fname,
                   u_int32 perm)
{
   path_id path;
   u_int32 count;
   u_int32 mode = S_IWRITE|S_IEXCL;
```

```
    errno = _os_create(fname, mode, &path, perm);
    if (errno!=0) return(errno);

    /* ok, got a file to write, how many bytes? */
    count = mod_ptr->_msize;

    /* now, write them to the file... */
    errno = _os_write(path, mod_ptr, &count);

    /* close the file... */
    _os_close(path);

    /* and return to the user */
    return(errno);
}
```

8.4.6 Fixing the Module CRC

Once the module has been saved to disk it can be loaded into memory for use at a future time, but only if the CRC is correct. When a module is created its CRC is made to be correct, but as soon as any of the data in the module changes, it is immediately incorrect. One way of restoring the CRC inside the file is with the fixmod command, seen earlier. Another is to execute the _os_setcrc() call prior to saving the module. This call takes a single parameter, the pointer to the module header, and updates the CRC value stored in the module.

8.4.7 Unlinking from a Module

When a link to a module is attained by either creating, linking, or loading, the link count on the module is incremented. Recall that a module will remain in memory until reaching a link count of zero, or negative one for sticky modules. The link count mechanism depends on modules being unlinked to work correctly, thus modules should be unlinked prior to the termination of a process that has linked to it. This is accomplished by calling _os_unlink(), passing in the pointer to the module header as the only parameter.

8.4.8 Crash Recovery using Data Modules

Another use for data modules is in implementing application crash recovery. Suppose you have an application which is started automatically when a system is booted and you have hardware which can detect system failures, cause a system reset and start the boot procedure. If your application needs to start up after a crash at or near the same place in code as when the last crash happened, this could be implemented by placing certain (or all) program global variables in a data module. From time to time, the data module could have its CRC updated and the module could then be saved to disk. Upon starting, the application would load this module from disk and have a copy of the last saved globals, allowing recovery.

If globals check-pointing is to be stored in non-volatile RAM, it is important to update the CRC on the module whenever data changes because, if upon reboot the module CRC is bad, the module will not be entered into the system module directory. However, performing an _os_setcrc() call is slow and calling this routine for every change to a global will slow the system down to a crawl. One way to speed things up would be to maintain two globals data modules, one in NV-RAM and the other in standard system space. The module in standard space would be used for active global variables, and periodically the data could be copied to the NV-RAM module followed by a CRC update.

CHAPTER 9

Interprocess Communication — Signals

This chapter and the next teach signals, alarms, events and semaphores — the tools OS-9 provides for synchronizing processes.

9.1 Signals

Signals are basically software generated interrupts. When a process has a signal sent to it, that process will (typically) put its current code location on hold, execute a special signal handling function and, upon completion of that function, continue executing at the place put on hold. Under OS-9, processes that do not have a signal handling routine are terminated upon receipt of a signal. (This is not the case for system state processes or signal 1.)

An OS-9 signal is a 16 bit number the handler routine accesses. Thus, different actions can take place determine

by the value of the incoming signal. Special signals and signal ranges are shown in Table 4.

TABLE 4 Defined Signal Values

Signal(s)	Definition
0	Unconditional KILL signal. User state processes receiving this signal are terminated.
1	Wake-up signal. Non-active processes receiving this signal are made active. No signal handler is necessary.
2	SIG_QUIT. This signal is sent when control-E is input on a serial device.
3	SIG_INT. This signal is sent when control-C is input on a serial device.
4	SIG_HUP. This signal is sent when a modem hang-up has been detected.
2 - 31	Deadly I/O range. These signals interrupt a process from performing an I/O action (reading or writing a device). Signals outside of this range will not be handled until the I/O is completed.
2 - 255	Reserved for current/future definition by Microware.
256 - 65535	User defined signals.

To successfully handle signals, a process must install a signal handling routine.

9.1.1 Signal Intercept Routine

The function _os_intercept() installs a routine, specified as one of the input parameters, as the process' signal handler. This routine takes two parameters: a pointer to the signal handler and a pointer to the process' static storage.

The installed routine is not designed to be called from within the program, rather to be called asynchronously by the OS-9 kernel on behalf of the process. Basically, when a signal is received by the process, the kernel changes the

process' program counter (PC) to point to the signal handler. Also, the process' registers are pushed onto the stack so the system can restore them later, and the register for the signal handlers first parameter is filled with the value of the signal causing the routine to be called.

Since the signal handler is called asynchronously, it must exit in a way that will restore the register set, including the PC. This is easily done by exiting the signal handler with the _os_rte() function. The handler routine must exit with the _os_rte() function or unpredictable results will occur.

Listing 9 contains a simple program designed only to receive signals and report their reception.

9.1.2 Sending a Signal to a Process

The _os_send() function will send the signal specified as the second parameter to the process specified as the first parameter. The unconditional kill signal (zero) can only be sent to a process owned by the same user/group as the sending process unless the sending process is a member of the super group (group zero). However, any other signal can be sent from any process to any process.

Listing 10 contains a simple program designed to take two parameters on the command line: a process id and signal value, and send the signal to the process.

If the recv program is running, the send program can be used to send signals to recv, and recv will report the signal reception. Notice that if signal one, the wake-up signal, is sent to a process, the signal handler is not called but the process is merely awakened. In this case, the recv program states it does not know the value of the signal that caused it to wake. Also, notice that if signal zero is sent to the recv program, it is terminated without reporting anything.

Listing 9. recv.c

```
/**** recv.c
** This program reports its process id and sleeps,
** waiting for signals.
**
****/
#include <stdio.h>
#include <signal.h>
#include <process.h>
#include <types.h>

/* extern void *_glob_data;    /* prior to Ultra C V1.2 */
#include <cglob.h>             /* beginning with V1.2 */

/* this next global must be declared "volatile", because */
/* it can (and will be) be changed in the signal handler.*/
/* Since Ultra C is a highly optimizing compiler, it might */
/* not allow these "external" modifications to the */
/* variable to be seen without the volatile keyword */
volatile signal_code LastSigRcvd;

void sighand(signal_code signal)
{
   LastSigRcvd = signal;
   _os_rte();
}

main()
{
   process_id myid;
   u_int16 mypriority, mygroup, myuser;
   u_int32 zero = 0;

   _os9_id(&myid, &mypriority, &mygroup, &myuser);
   printf("recv: process id %d\n",myid);

   / tell system about our signal handler */
   _os_intercept((void(*)())sighand, _glob_data);

   /* terminate program on signal 2 or 3 */
   while ((LastSigRcvd != 2)&&(LastSigRcvd != 3)){
      /* sleep until signal arrives */
      _os9_sleep(&zero);

      if (LastSigRcvd > 0) printf("Received signal
            %d\n",LastSigRcvd);
      else printf("Received signal ???\n");
      LastSigRcvd = 0;
   }
}
```

Listing 10. send.c

```
/**** send.c
** This program sends a signal input on the command
** line to the process
** specified on the command line.
**
**
** Usage: send <pid> <sigcode>
****/
#include <stdio.h>
#include <signal.h>
#include <process.h>
#include <types.h>

main(int argc, char *argv[])
{
   if (argc!=3){
      fprintf(stderr,"Usage: send <pid> <sigcode>\n");
      _os_exit(0);
   }

   _os_exit(_os_send(atoi(argv[1]), atoi(argv[2])));
}
```

9.1.3 Masking Signals

There are times in the execution of a program when it is inappropriate to jump to a signal handler, such as when speed is of the essence. To cause signals to line up and wait to be handled by the receiving process, the _os_sigmask() function can be used to build up a masking level and also take that mask down. All processes have a masking level which, if zero, allows signals to be handled and, if non-zero, causes signals to line up, in the order received, waiting to be handled. Since signals can be handled only one at a time, the system is prepared to stack up several signals on a process. As long as there is free memory on the system, signals can continue to build up when signals are masked.

_os_sigmask() takes an unsigned integer parameter which may be either 1, 0 or -1. A value of 1 adds one to the signal masking level, -1 subtracts one from the level and 0 brings the level to zero, allowing signals to be handled. In practice, the masking level should be incremented in areas which

want to run uninterrupted, decremented to clear up this incrementation when appropriate and only zeroed when signals should be handled immediately.

Listing 5 in Chapter 8 contains code to implement a timeout on reading from either a pipe or SCF device. As stated, the code will work most of the time but not always. The reason for this is it is possible the data could be ready at the time the _os_ss_sendsig() call is made, and if so, the signal will be sent prior to the process sleeping. Since _os_ss_sendsig() sends only one signal, the process will sleep for the entire timeout value even though a signal has been sent, and the process will then wake up and think it has timed out. The cure for this is to increment signal mask prior to installing the pending signal on the input device. (See Listing 11.)

Listing 11. timeout2.c

```
/* timeout2.c (just a subroutine)
*****************************************************
* The following program segment implements a simple
* timeout mechanism
* for receiving input from a pipe or serial device
* within a pre-defined
* amount of time. This version of code will work in
* all cases.
*****************************************************/

/*********
* Input parameters:
*   pn: the path number to get data from
*   buffer: the location to place data
*   count: the number of bytes to read
*   timeout: the number of seconds to wait for data
* Output Values:
*   0: timeout was reached before input arrived
*   #: the number of bytes read in by _os_read()
*********/
int readTimeout(int pn, int *buffer, int count,
                  int timeout)
{
   _os_sigmask(1);
   _os_ss_sendsig(pn,1000);
   _os9_sleep(&timeout);
   if (timeout) {
      /* we've reached timeout */
      _os_ss_relea(pn);
      return(0);
```

```
    } else {
        _os_read(pn, buffer, &count);
        return(count);
    }
}
```

After inspecting this code segment, it may appear that there is a bug since the signal mask is never decremented or implicitly set to zero. The question then arises, "Where should the _os_sigmask(0) call go?" (A value of -1 would not be acceptable because if signals were masked entering the routine, they would still be masked after the decrement.) Possible mask-clearing locations are just before the sendsig call, just after, and after the sleep function, but of course none of these do the trick. A mask clearing function is needed inside the sleep function. To avoid this problem, the sleep function does clear the signal mask at sleep time and, if a signal is already there, the process does not sleep.

9.1.4 Broadcast Signal Sending

On occasion it may be desirable to send a signal to many processes simultaneously. If the process id specified in the _os_send() function is zero, the signal specified will be sent to all processes on the system owned by the same group/user as the sending process, except the sending process.

9.2 Alarms

OS-9 alarms are merely signals that a process has requested to be sent to itself at some time in the future. Alarms differ from the signal sent as a result of an _os_ss_sendsig() call in that they are sent at a specific time. The time an alarm is to go off may be expressed as an absolute time and date or a certain number of clock ticks from the time of the request. An alarm can also be scheduled to repeat over a time interval expressed as clock ticks.

9.2.1 Alarms for a Certain Date and/or Time

An alarm may be installed to be sent at a specific time and date combination via the function called _os9_alarm_atdate(). This function takes the parameters: a pointer to a variable of type alarm_id, a signal_code, an integer specifying the time the alarm should be sent and another to dictate the date of the alarm. The time and date the alarm should be sent are coded 32 bit values as follows:

```
date: yyyymmdd
time: 00hhmmss
```

The location pointed to by the alarm_id pointer will return with a system id value for the installed alarm if all goes fine. This id is needed if the alarm is to be removed from the system prior to being sent. (Once a single-shot alarm is sent, it is removed.)

The function _os_alarm_atime() gives the software engineer another method of installing an alarm for which no date is specified. This call treats the time variable differently, however: time is the number of seconds that have elapsed since January 1, 1970, Greenwich Mean Time.

For those working with Julian time, there is another time/date based call available: _os9_alarm_atjul(). Time and date are available for specification with this routine. Time is input as the number of seconds after midnight the alarm should be sent and date is the Julian day number.

In all routines that install an alarm based on absolute time, the time may only be specified to the nearest second. However, the time of the alarm will adjust should the system's time and date be changed via the _os9_setime() function.

Listing 12 shows a program to install an alarm that will expire approximately five minutes from the start of the program.

Listing 12. sleep5minutes.c

```
/*** sleep5minutes.c ***/
#include <time.h>
#include <alarm.h>
#include <errno.h>
#include <types.h>
#include <stdio.h>

#define ALM_TIME 5*60       /* 5 * 60 seconds */
#define S_WAKE 1            /* wake-up signal */

main()
{
   time_t currtime, almtime;
   u_int32 ticks, zero=0;
   alarm_id aid;

   /* first, get the current time */
   _os_getime(&currtime, &ticks);

   system("date");
   printf("sleeping...\n");

   almtime = currtime + ALM_TIME;

   /* use S_WAKE signal so we don't need a signal handler! */
   if ((errno=_os_alarm_atime(&aid, S_WAKE,
                              (u_int32)almtime)) != 0){
      fprintf(stderr,"Error setting the alarm!\n");
      _os_exit(errno);
   }

   _os9_sleep(&zero);
   printf("Awakened at ");
   system("date");
}
```

Notice that this program does not have a signal handler. Since the wake-up signal is being used, one is not necessary.

9.2.2 Alarms to be Sent After Some Time Period has Elapsed

In real-time work it is not often necessary to have an alarm set up for an absolute time and/or date. The general need is for an alarm sent after some relative time difference from

the installation of the alarm. The call _os_alarm_set() takes as parameters: a pointer to the alarm_id, a signal_code and an integer which represents the number of system clock ticks to pass between alarm installation and execution.

The value for time is 32 bits, but only the lowest 31 bits are used to specify time. If the most significant bit is off, the alarm is installed to expire after the number of clock ticks (specified in the lower 31 bits) have expired. However, if the most significant bit is on, the lower 31 bits are taken as a number of 256^{ths} of a second that should elapse before the alarm is hit. Thus, a relative alarm may be installed to expire after a known amount of time without being aware of the system's time constant.

If a relative time alarm is installed, the number of ticks to pass before execution of the alarm must be at least two. This is because the system can only measure time to the nearest tick and if an alarm is scheduled for one tick and the next tick interrupt occurs immediately, the alarm will be sent without waiting any measurable amount of time.

Chapter 8 presented a program which used signals to implement a timeout condition if input on a device was not provided within some amount of time. If you closely analyze that program (or the updated version in Listing 11 in this chapter) you may realize that this timeout only works until the data starts arriving. If data starts but does not complete, the _os_read() function will, as is normal under OS-9's blocking I/O, halt until all of the data is received. To combat this situation we can install a relative alarm prior to the read request. This alarm will send a deadly I/O signal after some amount of time (our timeout period) and the routine needs to check the number of bytes actually read with the number of bytes attempted to read. (_os_read() updates the count variable with the number of bytes read; this is why a pointer to the count variable is needed.) The routine to timeout if input is not completed in a timely manner is shown in Listing 13. This routine abandons a timeout on data arriving but times out if input is not completed within the allotted

time. A better routine would be a combination of both methods, perhaps measuring the time taken until data begins to arrive (since the sleep function updates the sleep time variable to be the number of ticks remaining to sleep when interrupted) and timing out after the remaining time has expired if input is not complete. This routine will be left as an exercise to the reader!

Listing 13. timeout3.c

```
/* timeout3.c (just a subroutine)
*****************************************************
* The following program segment implements a simple
* timeout mechanism
* for receiving input from a pipe or serial device
* within a pre-defined
* amount of time.
*****************************************************/

/*********
* Input parameters:
*   pn: the path number to get data from
*   buffer: the location to place data
*   count: the number of bytes to read
*   timeout: the number of seconds to wait for data
* Output Values:
*   -#: number of bytes read but timeout occured
*   #: the number of bytes read in by _os_read(),
*   without a timout (successfully)
*********/
int readTimeout(int pn, int *buffer, int count,
                      int timeout)
{
   alarm_id aid;
   int numread = count;

   _os_alarm_set(&aid, DEADLY_IO_SIGNAL, timeout);
   _os_read(pn, buffer, &numread);
   if (numread != count) {
      /* the timeout was reached, indicate */
      /* with negative return value */
      return(-1*numread);
   } else {
      /* timeout was not reached, remove the alarm */
      _os_alarm_delete(aid);
      return(count);
   }
}
```

9.2.3 Cyclic (Repetitive) Alarms

Some programs need to receive a signal every X clock ticks as either a heartbeat or perhaps a cue to perform a task. This is set up by using a cyclic alarm - one that reschedules itself with every alarm period expiration. (If you have a cyclic alarm set up to send a signal once each second, on the second, there will be no skew from one signal to the next.) The function to install a cyclic alarm is _os_alarm_cycle(). Its input parameters parallel those of _os_alarm_set(), and its operation differs only in that this is not a single-shot alarm.

Cyclic alarms are useful for any function needing to be

9.2.4 Removing Alarms from the System

In the case of either a cyclic alarm that is no longer needed or a single-shot alarm that has not yet arrived, a program may wish to remove an alarm. This can be performed as long as the process has maintained the system id number for that alarm, given to the process at the time of the alarms installation. The function _os_alarm_delete() takes the id number as its only parameter and removes that alarm from the system. An example of this function is seen in Listing 13 above.

CHAPTER 10

Interprocess Communication — Semaphores

Semaphores provide a method of synchronizing processes so they can perform some coordinated task. This chapter discusses the various methods available for implementing them.

10.1 Semaphores

A semaphore is a simple binary value typically used to control access to resources to which only one entity (process) may have access at any given time. Basically a semaphore is a token which a process must own to access the resource. When a process needs that resource, it competes for the token (semaphore) and, when received, it uses the resource. When the process is done with the resource, it returns the token so other processes wanting that resource can get it.

10.1.1 Creating the Semaphore

Before a semaphore can be used, the system must be aware of its presence. The creating process must allocate a 32 byte area of memory for the semaphore and tell the system where this area is. Generally, this area will be located inside a data module allowing other processes to find the starting

address as an offset from the data module's data area. This saves the program from requiring a hard-coded location for the semaphore.

The routine to create a semaphore is _os_sema_init(). Its only parameter is the starting address of the semaphore's memory. Each application should call this function prior to using the semaphore.

10.1.2 Competing for the Semaphore

To use the resource controlled by the semaphore, it must be successfully competed for through the _os_sema_p() call. Once the semaphore is owned, access to the resource is granted. This call does not return until the current holder of the semaphore (and all other requestors before our call) releases it. The only parameter used is the pointer to the semaphore structure.

10.1.3 Releasing the Semaphore

When the process owning a resource is through using it, that process should return the semaphore token, or no other processes will be able to gain access. Returning the semaphore is as simple as a call to _os_sema_v(). The only parameter used is the pointer to the semaphore structure.

10.1.4 Removing the Semaphore from the System

Once a process is through using a semaphore, it should inform the system since so the system may remove it. The call _os_sema_term() unlinks the calling process from the semaphore. The only parameter used is the pointer to the semaphore structure.

10.1.5 Sample Code Using a Semaphore

This application will create a data module to place a semaphore structure inside of it, initialize the semaphore and cycle through a loop ten times. The application will 1) com-

pete for the semaphore, 2) sleep 10 seconds and 3) release the semaphore. Once the loop is fallen through, the process terminates use of the semaphore. Since there is a sleep period between each acquisition and release, there is sufficient time to start multiple processes running the same application. (If you run multiple process of this application in one window/terminal, you can differentiate the output generated by observing the id numbers that are also output.)

Listing 14. sememo.c

```
/*** sememo.c ***/
#include <stdio.h>
#include <semaphore.h>
#include <module.h>
#include <types.h>
#include <errno.h>

/* this value will hold our process id */
/* extern _procid;    /* prior to Ultra C V1.2 */
#include <cglob.h>    /* beginning with V1.2 */

#define DMOD_NAME "dm_sema"
#define LOOP_COUNT 10

main()
{
   u_int16 attr_rev, type_lang;
   u_int16 perm, dm_size;
   mh_com mod_head;
   Semaphore myflag;
   char *d_name;
   u_int32 sleeptime, x;

   dm_size = sizeof(semaphore);
   type_lang = (MT_DATA<<8);
   attr_rev = (MA_REENT<<8);

   perm = MP_OWNER_READ|MP_OWNER_WRITE;

   errno = _os_datmod(DMOD_NAME, dm_size, &attr_rev,
      &type_lang, perm, (void**)&myflag,
      (mh_data**)&mod_head);

   /* if the creation failed, maybe the */
   /* module already exists */
   if (errno!=0) {
      if (errno != E_KWNMOD) {
         fprintf(stderr,"%d: Couldn't make the
            data module!\n",_procid);
```

```
            _os_exit(errno);
         }
         d_name = DMOD_NAME;
         errno = _os_link(&d_name, (mh_com*)&mod_head,
            (void**)&myflag, &type_lang, &attr_rev);

         if (errno!=0) {
            fprintf(stderr,"%d: Couldn't link to the
               data module!\n",_procid);
            _os_exit(errno);
         }
      }

      errno = _os_sema_init(myflag);
      if (errno != 0) {
         fprintf(stderr,"Error initializing
            semaphore\n");
         _os_exit(errno);
      }

      printf("%d: I have access to the semaphore
         module\n",_procid);

      for (x = 0; x < LOOP_COUNT; x++) {
         printf("%d: Competing for the semaphore...\n",
            _procid);
         _os_sema_p(myflag);
         printf("%d: I've got the semaphore...\n",
            _procid);

         /* sleep for 10 seconds */
         sleeptime = (10 << 8) | 0x80000000;
         _os9_sleep(&sleeptime);
         /* note, without the delay, our process might */
         /* get the semaphore again, even though another */
         /* process is waiting! */

         printf("%d: Releasing the semaphore...\n",
            _procid);
         _os_sema_v(myflag);
      }

      printf("%d: Program is exiting normally...\n",
         _procid);
      _os_sema_term(myflag);
      _os_unlink(mod_head);
      _os_exit(0);
   }
```

10.2 Events

Events are a very powerful feature provided by OS-9 to implement semaphores, predating the implementation of semaphores described in Section 10.1. An OS-9 event differs from the traditional semaphore in that the latter can typically take on one of two values, on or off. The former contains a 32 bit signed value which processes can manipulate to a very large extent.

On a basic level, a process can either wait for an event to occur or signal the fact that an event has occurred. Since internally an event is a signed integer, this means that a process can wait for an event to be a specific value (or in some range of values) or change the value. There are more things that can be done with events, as we shall explore.

10.2.1 Creating a Non-Existing Event

Before an event can be used, it must be created. This is performed via the _os_ev_creat() call, which takes as parameters: the name of the event to create, a "Winc," a "Sinc," a value to which the event will be initialized and a pointer to store the event_id which will be used to perform other actions with the event. A list of all events currently on the system may be received through the events call at the shell. (Winc and Sinc are explained in sections 10.2.3 and 10.2.4 respectively.)

10.2.2 Linking to an Event Already Created

If an event is already present on the system, it must be linked to before it can be used by a new process. This is accomplished through the _os_ev_link() call, which takes the name of the event and a pointer to store the id value.

10.2.3 Waiting for an Event to Occur

One of the basic actions performed with an event is to wait for it to occur. For example, a process may need to wait

until three other processes have signaled the event, bringing the value of the event to three (assuming the event started at zero and was incremented by one for each event signal). The process that needs to wait will execute an _os9_ev_wait() call, which takes as parameters the event's id, a location to store the value upon returning from the call, a pointer to store the signal value causing premature return from the call, a minimum acceptable value and a maximum acceptable value. In this case the process would wait for values between three and three (indicating that only value three is acceptable to return from the wait call).

Recall that one of the parameters to the create call is called Winc. This value is a wait increment; that is, a value to be added to the event's value after a process has successfully waited for that event. (If this feature is not needed, the Winc value can be set to zero.) Once Winc is set up (at event creation time) it may not be changed.

➤ If a process returns from waiting prematurely due to receipt of a signal, the Winc value will not be applied to the value of the event. Also, under OS-9000, the _os_ev_wait() function should be used to wait. This call includes as a parameter, a pointer to the location that the value of the signal (that caused the program to return prematurely) should be placed.

10.2.4 Signaling that an Event has Occurred

In the small example presented in Section 10.2.3, there are three other processes which must each increment the event value by one, bringing the event from an initial value of zero to a final value of three. Each process could make the call _os_ev_signal(), passing the appropriate event_id, and a parameter called AllFlag (to be explained later). The parameter called Sinc in the creation call is a signal increment — a value to be added to the event's value whenever it is signaled. Thus, if the Sinc were set to one (Sinc also cannot be changed after creating the event), then each process would increment the event's value by one, and when each of the three processes had signaled the event, its

value would reach three, allowing the process waiting for three to continue.

AllFlag is a 16 bit boolean value, considered to be true if the most significant bit is on, making a hex value 0x8000 or greater. (The remaining bits are currently undefined, but must be zero.) If AllFlag is true for any call containing it as a parameter, the order of operation after the call is:

1. change the value (i.e., add Sinc to the value)
2. wake the first (next) waiting process this new value satisfies
3. add the wait increment if a process was satisfied
4. if a process was satisfied goto step 2.
5. return to caller

If, however, AllFlag is *false*, the system will skip step 4. and the goto loop.

➤ Note: If an event's value changes as a direct result of waiting for a value range which includes the current event value (i.e., no event signal/set call needs to occur), the Winc will be applied but no other waiting processes will wake up due to this new value. This is because _os9_ev_wait() does not contain an AllFlag parameter.

10.2.5 Reading the Value of an Event

The value of an event can be read through the _os_ev_read() call. This call takes as parameters the id of the event, and a pointer to the storage location of the value.

10.2.6 Setting the Value of an Event

The value of an event can be set to an arbitrary value with the _os_ev_set() call. Parameters supplied are the id of the event, a pointer to the value the event should be set to and AllFlag. The value of the event prior to the change will be placed at the location of the new value.

10.2.7 Pulsing an Event

Pulsing an event is an interesting consideration. To pulse an event is to:

1. store the current event value
2. set the event to a value specified as a parameter to the pulse call
3. wake the first (next) waiting processes if this value satisfies it
4. add Winc
5. if AllFlag is true, go back to step 3.; else continue to step 6.
6. restore the original event value recorded in step 1.

Pulsing is something that does not come up very often but occasionally is very useful. For example, assume your imaging analysis application has five processes: a red process that performs its work on the red frame information, green and blue processes that each do similar work on the green and blue frames respectively, a composite process that performs its work no matter what color frame is available and a master process that is responsible for gathering each color frame, one color at a time, and alerting the proper color process as well as the composite process that data is available. One method of getting this task done would be to signal the composite and color processes with each frame of data, but a more simple method would be to use events and pulsing. Lets specify a Sinc of zero and a Winc also of zero, as is often the case when events are being pulsed. (Generally, when an event is used via pulsing, it is not signaled, and visa-versa.) Each of the color/composite processes would have a loop similar to the pseudo-code below:

Listing 15. partial_pulse_code.c

```
/*** partial_pulse_code.c ***/
#define RED       10
#define BLUE      20
#define GREEN     30
#ifdef I_AM_RED
#  define WAIT_PARAMS      RED,RED
#else
#  ifdef I_AM_BLUE
```

```
#     define WAIT_PARAMS      BLUE,BLUE
#  else
#     ifdef I_AM_GREEN
#        define WAIT_PARAMS      GREEN,GREEN
#     else
         /* must be I_AM_COMPOSITE */
#        define WAIT_PARAMS      RED,GREEN
#     endif
#  endif
#endif

workloop()
{
   /* event has been linked to with global */
   /* variable eid as the event_id */
   while (!timeToQuit){
      /* wait until my frame is available */
      _os_ev_wait(eid,&ev_value,&sigcode,WAIT_PARAMS);
      DoMyFrameWork(); /* this routine does the work */
   }
}
```

Looking at this code, notice toward the top a definition called WAIT_PARAMS which is defined as the appropriate parameters to the event wait call with respect to one of the color processes. Each of red, blue and green wait for exactly one value, while composite waits for any value between red and green; the master process will pulse the appropriate color value upon receipt of that color of data. Thus, for each frame the appropriate color process *and* the composite process will wake up from the wait call upon receipt of that color of data. (For this to work, the master process must set AllFlag to true for the pulse call, otherwise only one of the two processes will wake. Also, of course, the composite process must return to the top of the loop before the next frame arrives.)

Not seen in the code is the call to pulse an event: _os_ev_pulse(). Parameters to this call are the id of the event, a pointer to the value of the event and AllFlag. The pointer is also used to place the value of the event before (and after) the pulse.

10.2.8 Accessing an Event on Relative Terms

An event may be waited for or set based on relative values (that is, treating the current value as if it were zero). The call _os9_ev_waitr() is very similar to _os9_ev_wait() but the minimum and maximum parameters are taken to be offsets from the current event value. Also, _os_ev_setr() allows an arbitrary value to be added to the current event value. Each of these operations could be done by performing an _os_ev_read() followed by the non-relative version of the above routines (basing parameters as offsets from the current value), but these routines allow for quicker operation (by requiring fewer steps).

➤ A compelling reason to use the relative calls to wait and change the event is the atomic nature of them. We could first read the event and then take some action with it, but we have no way to stop other processes from changing the value between our operations. Relative calls are performed in one step.

10.2.9 Unlinking from an Event

When an event is no longer needed, it should be unlinked. Events maintain a link (or usage) count informing the system how many users are using (linked to) each event. (When an event is no longer being used, it can be removed from the system.) A process is not automatically unlinked from an event when it terminates, so a process that links to an event (or creates an event) should make a call to _os_ev_unlink() passing in the id of the event once that event is no longer needed.

10.2.10 Removing an Event from the System

_os_ev_delete() is used to remove an event from the system when it is no longer needed. This call takes as its only parameter a pointer to the name of the event to be removed. An event may only be removed if the link count for the event is zero, indicating that no processes are using the event.

CHAPTER 11

Process Scheduling

This chapter explains how the operating system decides what process should run and for how long.

As mentioned earlier, OS-9 supports two types of processes: those that run in user state and those that run in system state. When a system state process starts running, it keeps the CPU until it wishes to relinquish (or until an interrupt comes in) and other processes will not run. (This is now up to the system state application designer, since there is a system global which decides whether or not the system can preempt system state code.) However, the normal process is a user state process which gets a certain amount of time in the CPU before the kernel allows another process to run. The amount of time a process is permitted to run before the kernel considers other processes is called a time slice. This amount of time is usually two ticks (the time it takes for two interrupts to be generated by the system clock), but this is changeable by either modifying the init module and rebooting, or by altering the d_tslice system global. (For more information on system global variables, see Chapter 16.)

11.1 Active Queue

To earn the right to use the CPU, processes must move to the front of a line called the active queue. The active queue is a list containing all processes that are ready for CPU time, constantly sorted in the order of the next process to enter the CPU, to the last. When the kernel needs to run a new process, it takes the process at the front of the active queue.

The priority of a process has a great deal to do with how often it wins CPU time. (The absolute value of a process' priority is meaningless; it is the relative priority as compared with other active processes' priorities that makes the difference.) When a process enters the active queue, its initial position is directly dependent on the process' priority. The kernel compares the entering process' priority with the priority of each process already in the queue, from the back of the queue toward the front. As long as the new process' priority is greater than the priority it is being compared to, the process continues to move toward the front of the queue. When (if) the process meets another with a priority equal to or greater than its own, it must enter the queue behind this other process. This form of process scheduling is a combination of priority scheduling (the process with the highest priority gets to run) and round-robin scheduling (a fair arbiter which allows all processes to get a turn). OS-9's process scheduling goes one step beyond.

➤ The actual algorithms used for insertion sorting and process aging are different as described above and below. They are more complicated than the discussion may lead you to believe, but are much faster as a result, and the outcome of process position is the same as the simple algorithms presented would yield.

Since the form of scheduling already mentioned could potentially leave processes locked out of the CPU, OS-9 includes a third form of scheduling called aging which treats priority differently. When a process enters the active queue, for instance, rather than compare process priorities as above, the kernel compares process ages for placement

into the queue. Priority is still very important, however, because the age of a process is set equal to its priority whenever it enters the queue. Every time the kernel undergoes a context switch (changes the running process from one process to another) some process will enter the CPU while perhaps others will stay behind in the active queue. The processes that are left behind are permitted to increase in age by one and eventually all processes will get their fair share of execution time.

Assume a system has three processes, process A with a priority of 130 and processes B and C each with a priority of 128. Table 5 shows the first twelve iterations (plus a set up iteration where no process runs) of the active queue and CPU. For a simple scenario involving these processes, assume that each process wants as much CPU time as it can

TABLE 5 Process Scheduling: A Simple Case

Iteration	CPU	Active Queue		
pre	<none>	A_{130}	B_{128}	C_{128}
1	A^{130}	A_{130}	B_{129}	C_{129}
2	A^{130}	B_{130}	C_{130}	A_{130}
3	B^{128}	C_{131}	A_{131}	B_{128}
4	C^{128}	A_{132}	B_{129}	C_{128}
5	A^{130}	B_{130}	A_{130}	C_{129}
6	B^{128}	A_{131}	C_{130}	B_{128}
7	A^{130}	C_{131}	A_{130}	B_{129}
8	C^{128}	A_{131}	B_{130}	C_{128}
9	A^{130}	B_{131}	A_{130}	C_{129}
10	B^{128}	A_{131}	C_{130}	B_{128}
11	A^{130}	C_{131}	A_{130}	B_{129}
12	C^{128}	A_{131}	B_{130}	C_{128}

get. The pre-iteration shows the active queue when all three processes have lined up, assuming that process B enters the

active queue before process C. Based on this active queue, process A enters the CPU and each of the remaining processes increases in age by one. After the time slice for process A expires, it is momentarily reentered into the active queue, also shown as part of iteration 1. Notice that the age of B and C is now 129 and the age of A is 130, enough to move it straight to the head of the queue for the next iteration.

In the second iteration, when process A leaves the CPU, B and C have each aged to 130. Because of this, A is positioned at the end of the queue when it leaves the CPU.

After the first four iterations have expired, we will begin to see a pattern in the CPU cycles that follow. Starting with iteration number 5, the cycle is A, B, A, C and it then repeats. No matter how many iterations are computed, the pattern of processes in the CPU will remain consistent as long as all other things remain static. The pattern generated with this arrangement of processes and priorities contained 4 iterations, of which process A runs for 2 iterations while processes B and C each run for 1 iteration; thus, process A commands the CPU for $2/4^{ths}$ (50%) of the time while processes B and C each control the CPU for $1/4^{th}$ (25%) of the time.

With the understanding of how to compute percentages of CPU time across processes with various priorities, it may be tempting to compute tables similar to above in an effort to devise just the right priority ratios for your system. This effort, as exciting as it may be, iS basically useless. The percentages computed are only accurate when processes involved want the CPU as much as they can get it. This assumption is not a realistic case, however, as processes regularly spend time that does not involve running in the CPU. Some examples of these periods include the time a process sleeps, waits for child processes to complete, waits for events or semaphores, or block performing device I/O.

11.2 Sleep Queue

The sleep queue is a holding location for processes that have executed the _os_sleep() instruction. This call takes as a parameter a number indicating how long the process will remain in the sleep queue, or zero to indicate the process will sleep forever (a sleep for no specific amount of time). Processes in the sleep queue for a timed sleep (non-zero parameter) will leave the sleep queue once their sleep time has elapsed, or sooner if the process receives a signal. (Sleeping clears the signal mask of a process if there is one.)

To make life simple for the kernel, processes in the sleep queue are sorted in order of the next process to wake up to the last of the timed sleeps, and untimed sleeps are placed behind these. In this manner, when the system's clock interrupt causes the kernel's interrupt routine to wake the system process, the system needs only to search the beginning of the sleep queue looking for processes to wake. As soon as the system finds a process whose wake-up call is not ready, it is assured that none of the processes that follow are ready to awaken either. Incidentally, when a process wakes of natural causes (no signal caused the wake-up), it is placed into the active queue with a position dependent upon its age as described above.

11.3 Wait Queue

When a process that has created child processes has nothing else to do (at least for the moment) it might wait for one of its child processes to terminate before executing again. For example, the shell program will wait for its child process to terminate before sending another input prompt to the terminal, unless the ampersand operator is placed on the input line to specify background operation. When a process is waiting for a child to terminate, it is placed in the system's wait queue, where it will remain until one of its child processes terminates. This queue is not sorted in any particular order.

A signal will remove a process from the wait queue if the signal mask is cleared.

11.4 Event Queue

Each event on the system has a queue of its own to hold processes that are waiting for their value to come into some range. The order of processes in an event queue is first come, first served, without regard for process priority. Thus, if several processes are waiting for a certain value to be set, the first waiting process will wake when the value hits the magic number, and if Winc has been set to a non-zero increment, other processes will not wake up at the same time.

A signal will remove a process from an event queue if the signal mask is cleared.

11.5 Semaphore Queue

Just as for events, each semaphore on the system has it's own queue of waiting processes. When a semaphore is released (v-ed), if the semaphore structure indicates waiting processes, a system call will unblock all waiting processes. Then, all of the unblocked processes will attempt a test-and-set operation on the semaphore, and the first process (generally, the one with the highest priority) will own the semaphore while the other processes will re-block.

11.6 Device I/O Queue

When a process enters a device driver, the driver is marked as being busy. If other processes attempt to use the same device, the operating system will cause them to enter a queue of processes waiting to perform I/O on that device. The position inside the device's queue is dependent upon its priority (and subsequently age).

A deadly I/O signal will remove a process from a device's I/O queue. For more information on the deadly I/O signals, see Table 4 in Chapter 9.

11.7 Process Preemption

Preemption is defined as the action of one process forcing another from the CPU before the time slice of the process in the CPU has expired. Under OS-9, only a relatively high priority process may pre-empt a relatively low priority process, and only if the process doing the preemption is not in the active queue at the time.

To preempt another process (process A), the process attempting preemption (process B) must receive a signal causing the process to be removed from either the sleep, wait or event queue, or a device I/O queue. If process B's priority is not greater than that of process A, process B will be inserted into the active queue to wherever its age (set equal to its priority) dictates. (No special consideration moves process B to the head of the active queue.) However, if process B's priority does exceed that of process A, process A is placed back into the active queue (using the normal mechanism) and process B begins to run. At this point, normal process scheduling continues.

CHAPTER 12

Combining Assembly Language with C Routines

There are many reasons a software engineer may wish to combine code written in C with code written in assembly language in one project. This chapter covers combining assembly and C languages.

If these two languages are to be married, the code needs to be able to communicate. This is accomplished quite easily provided the engineer has a firm understanding of the register convention used by the C compiler. Table 6 shows this convention.

12.1 Hardware Information

The CPUs in the Motorola 68k family all have eight general purpose data registers and eight general purpose address registers, named d0 through d7 and a0 through a7, respectively. A7 is also referred to as sp, since this register is not

really general purpose but is used as the stack pointer. All of these registers are 32 bits wide.

TABLE 6 Register Convention of Microware's C Compiler

Register(s)	Usage
d0, d1	These registers are used to hold the first two parameters in a subroutine call.
d0	This register is also used to hold the return value of a subroutine.
d2, d3	These registers are used by the compiler to hold intermediate results in complicated expressions.
d4 - d7	Register class variables are located in these registers.
a0, a1	These registers also hold intermediate results, this time for pointer arithmetic.
a2 - a4	Register class pointers are stored here.
a5	This is the stack frame pointer.
a6	This is the global variable pointer.
a7	This is the stack pointer.

12.2 Subroutine Parameters

The first two parameters to subroutines are generally placed in d0 and d1. If there are more than two parameters, additional parameters are placed on the stack in the order of last parameter to third. (This ordering allows for optional parameters if the compiler has not been given a function prototype dictating parameters.)

If the second parameter to a function is larger than 32 bits, it will be placed entirely on the stack and d1 will not be used. If the first parameter is larger than 32 bits but smaller than (or equal to) 64 bits, it will be placed across d0 and d1 and the second parameter will be on the stack.

12.3 Register Class Variables

A register class variable is one that has been defined with the register keyword before its type. These variables are placed in a register rather than on the stack, yielding significantly quicker access times.

Ultra C, the current C compiler under OS-9, does not expect the programmer to implicitly declare any variables as registers. In fact, this compiler will ignore these requests and determine for itself which variables are best suited for register placement.

12.4 Stack Frame Pointer

For those not familiar with the stack frame pointer concept, this section should be very interesting and enlightening. As any experienced assembly programmer can attest, when a subroutine exits, the stack should contain the same amount of information as when the routine started. Without higher level support, this means the routine must keep track of the number of bytes pushed onto the stack so they can all be removed later. We will see that the stack frame pointer greatly alleviates this problem.

The link instruction of the 68k instruction set takes two parameters: a register to serve as the stack frame pointer and a size that will be explained later. Using register a5 as our stack frame pointer, the link instruction performs the following:

1. Push the current value of the a5 register onto the stack
2. Make a5 point to the location on the stack that the previous value was placed
3. Add the size parameter to the stack pointer

There is a counter instruction, unlk, which reverses what the link instruction does, restoring the a5 register to its original contents and also restoring the stack pointer to its value prior to the link.

Usually, when a routine is entered the link instruction is called, allowing the routine to use the stack for any temporary storage necessary. Just before exiting, the routine calls unlk to restore the stack and return that temporary memory.

The stack is also used for holding local variables while the routine is executing. Forgetting that we have a stack frame pointer, the routine must access these variables based on the stack pointer which is moving (with every push or pop). Using the link instruction to provide a stack frame pointer provides a fixed reference with which to access these local variables. This is where the size parameter comes in. Assuming the routine needs 16 bytes of stack space for local variables (4 integers, perhaps), the size should be specified as -16. (Remember, this size is *added* to the stack pointer after the register push. Since the stack starts high in memory and works its way down toward lower addresses, a negative value must be used or stack corruption will result.)

Listing 16 shows a simple routine which takes a single integer parameter and adds one to it.

Listing 16. Simple Assembly Routine

```
* Simple Assembly Routine example code
* an asterisk in column one means rest of line
* is a comment!
* this routine needs to be surrounded by other code
* to make it through the assembler

* This subroutine could be prototyped for C like:
* int AddOne(int);
* and could be used like:
*
*  int A=5, B;
*  B = AddOne(A);
*  /* now, B == 6 */
AddOne:
   link a5,#0      * make a stack-frame pointer
   addq.l #1,d0    * add one to the d0 register
   unlk a5         * restore stack and frame ptr
   rts
```

In the previous example, the stack frame pointer was really not necessary but was included to show the programming

technique. Listing 17 shows an example using a local variable referenced from the stack frame pointer.

Listing 17. Assembly with Local Variables

```
* another assembly example, with variables
*  What this routine emulates:
*  int SomeFuction(int a, int b, int c, int d)
*  /*c and d will be parameters on the stack */
*  {
*     int x=10;       /* local variable on the stack */
*     /* yeah, I know, this is a trivial */
*     /* example, but it is all I could think of! */
*     return(x+a+b+c+d);
*  }
SomeFunction:
   link a5,#-4     * leave space for 4 bytes
* now, the stack looks like this, with each entry
* being four bytes large:
*
*     +-------------+
*     | parameter d |
*     +-------------+
*     | parameter c |
*     +-------------+
*     | return addr | (put here by bsr instruction)
*     +-------------+
*     | old a5 reg  | <--- A5 (put here by link)
*     +-------------+
*     | local var x | <--- SP
*     +-------------+
* save register d2 for temporary usage
   move.l d2,-(sp)
* initialize local x to 10
   move.l #10,-4(a5)
* move local x into d2
   move.l -4(a5),d2
* add d2 and d0 (a) and put result in d0
   add.l d2,d0
* move parameter c into d2
   move.l 8(a5),d2
* add it into d0
   add.l d2,d0
* move parameter d into d2
   move.l 12(a5),d2
* add it into d0
   add.l d2,d0
* and finally, add d1 into d0
   add.l d1,d0
* now, restore d2
   move.l (sp)+,d2
* and restore the rest of the stack and return
```

```
* (d0 already contains the return value)
   unlk a5
   rts
```

➤ The example just presented is not meant to be shown as an optimal design, instead to show the register to variable interaction. If the Ultra C compiler were to compile the C routine listed in the assembly comments above, the routine would be much more optimized and perhaps more readable as well.

12.5 Global Variable Pointer

The a6 register is used as the pointer to a process' global variables. For historic purposes this register will be initialized to point to the base of global variables plus 0x8000. This allows the 68000 to access 64k of globals directly using a6 as a base offset. The code in Listing 18 contains a routine that will call the F$Sleep kernel service in a manner similar to the standard routine _os9_sleep(), with a small difference. The standard C routine will return any error number encountered, while this new routine returns -1 in the case of an error and places the error code in the global variable called errno.

➤ Notice the F$Sleep function being specified in the code as a parameter to the os9 function. As much as Microware would have liked that instruction to be included in the 68k instruction set, this is not the case. Instead, Microware's assemblers recognize this instruction as a macro, which causes a trap to occur on trap vector zero. All processes under OS-9 have the system kernel as a pseudo trap handler which is called for vector zero traps. For more information on trap handlers, see Chapter .

Listing 18. New sleep() function

```
* my_sleep.a (code only, no psect)
*
* int my_sleep(int x)
* sleeps for the number of ticks passed
* as parameter x, returns negative 1 if
* an error occurs or the number of ticks
* remaining to sleep when awakened.
```

```
*
* for function details see the manual entry
* for the F$Sleep routine.
my_sleep:
   link  a5,#0     * good habit to get into
* d0 is already set up as parameter to F$Sleep
   os9   F$Sleep
   bcc   sleep_had_no_error
* error if we reach here
* prepare to return -1
   move.l #-1,d0
* and put error code in errno global
   move.w d1,errno(a6)

sleep_had_no_error
   unlk a5
   rts
```

Notice that in the above example, referring to the global errno was as simple as specifying the name of the global referenced from the global variable pointer, a6. The linker stage of the compiler will take care of converting this name to the number of bytes the variable is placed away from the data pointer when building the machine code.

12.6 Placing Assembly Directly within a C Code Module

A special function is built into the Ultra C compiler to allow assembly instructions to be placed within a C source file; that instruction is _asm(). It takes a minimum of one parameter, a constant character string representing the line of assembly. (Additional parameters may be placed beyond the assembly string if referred to within the string. See the compiler manual for details.) This instruction should not be used within the local context of any C subroutine, however, because the optimization phases of the compiler may alter the flow of instructions in a non-intuitive way, causing your additions to damage what has been done. Since the compiler does not optimize assembly code, if you embed assembly directly within a C routine, your code may easily crash. Listing 19 contains an example which uses the routine presented in Listing 16.

Listing 19. Embedded Assembly within C

```
/*** asm_in_c.c ***/
#include <stdio.h>
int AddOne(int);

main()
{
   int A = 5, B;

   printf("Before subroutine call, A=%d, B=%d\n",
      A,B);

   B = AddOne(A);

   printf("After subroutine call, A=%d, B=%d\n",
      A,B);
}

_asm("AddOne:");
_asm("   link a5,#0");
_asm("   addq.l #1,d0");
_asm("   unlk a5");
_asm("   rts");
```

➤ Beginning with Ultra C Version 1.2, _asm() calls may be placed within C functions, allowing you to directly manipulate the code generated. To allow this and still allow optimization, you must write not in direct assembly, but instead in a macro-type assembly, which Ultra C uses to determine your intentions. The details on this macro language were not available with the first printing of the primer.

CHAPTER 13

Locating and Executing Code External to the Main Program Module

This chapter is dedicated to subroutine modules and trap handlers, the two methods OS-9 has available for using shared libraries.

13.1 Subroutine Modules

A subroutine module is a module which holds code to be executed by processes whose primary module is somewhere else. Simply put, it is a shareable library. This library does not have to be linked to the main program, but is instead attached to during the main program's execution. During execution, the client (calling program) will link to the subroutine module, read an index table pointing to functions and assign local function pointers to point directly to these routines. Since the pointers are for absolute locations rather than relative, they do not have to be set at program link time.

Subroutines located within a subroutine module may *not*, under any normal circumstances, access global variables (by name) that are either declared within the library itself or the calling programs. Since references to a variable must be resolved at link-time (and subroutine modules are linked

separately from the applications that use them), the linker will fail if the routine names a global of the callers. Additionally, though it is possible to define globals in the code for a subroutine module, since these modules do not have any static storage area prepared for them by the system kernel, using a global declared like this will corrupt memory.

➤ In the OS-9000 operating system, OS handling of subroutine modules is somewhat different, more resembling trap modules. One primary difference is that subroutine modules may have their own static storage areas.

The use of a subroutine module is very simple, requiring minimal extra code or variables in the program. The creation of a subroutine module is also simple, requiring one short code file to tie things together.

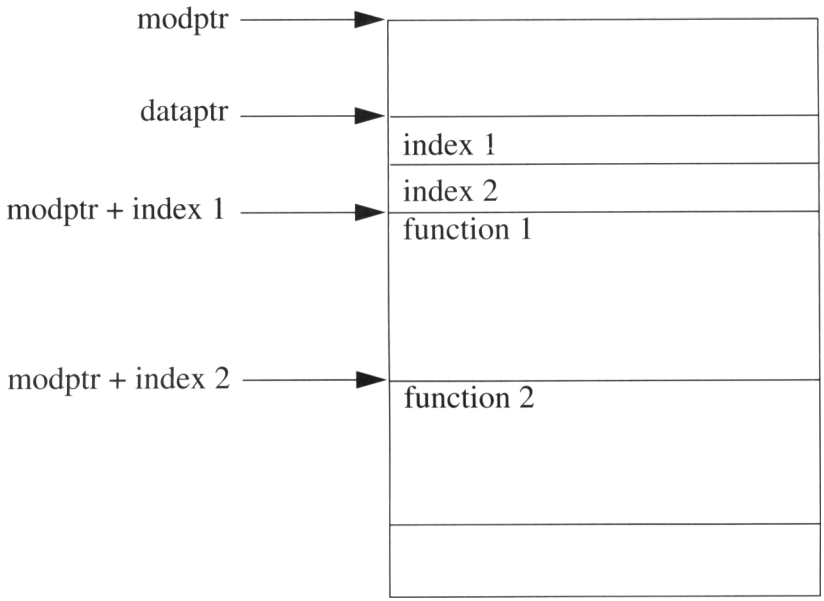
13.1.1 Coding a Subroutine Module

From the module standpoint, a subroutine module is like any program module except it has no initial starting point. (All program modules have an initial starting point so that when executed, the system knows where to begin the process.) Actually, subroutine modules do have a starting point, but we will see that it is used differently.

When a program attaches to a subroutine module it must have some method of finding the callable routines. The standard for doing this easily is to place an index table in the module. For best results, this index table should be pointed to by the _mexec offset within the module header. (This is the dataptr parameter returned by the _os_link() function.) This index table is composed of offsets from the beginning of the module to the routine to which each entry corresponds. (It is the responsibility of the software engineer to make sure the client program is aware of the order of index entries.) Client software must then add each offset to the beginning of the module and the result is the absolute location of the routine.

➤ There is no need to terminate the index table in any special way since the client routine should know in advance how many routines should be listed. (The system does not prepare anything for the client.) If more routines are added to a subroutine module in a later update, clients that had been written for the previous version (and number of routines) will continue to operate correctly provided the order of the original routines has not changed in the index table. (Of course, the old client software will not know that new routines have been added. Even if it could discover this, it probably would not know what the new routines are for or what parameter convention they require, without some very tricky coding.)

FIGURE 21 Location of Code within a Subroutine Module



A normal C program has as its execution entry point a routine called _cstart, (pointed to by the _mexec structure element located in the module's header), which initializes some global variables and calls the main() routine. Since subroutine modules don't need _cstart but do need an index table, rather than allow the subroutine module to be linked with cstart.r, we need to build our index table in assembly language and use it as a replacement to cstart.r. This index table will be written so the _mexec offset will point to it and

allow client software to easily find it. The actual procedure will mimic that of getting a pointer to the data within a data module, but this data will merely be an undetermined number of integers. (With this format the client needs to merely use an integer pointer as the *dataptr* and read the values from this pointer as if it were an integer array.)

13.1.2 The Index Table

The index table of a subroutine module is a simple array of integers, each being an offset from the beginning of the module header to the subroutine to which the index entry corresponds. The code to build an index table is written in assembly allowing the table to be positioned first inside the body of the module. An example is shown in Listing 20.

➤ In an assembly file, any item beginning in the left-hand column is taken to be a location label while code mnemonics must *not* begin in the left-hand column.

The index table is put together toward the bottom of the listing. It contains three entries corresponding to functions: Func1(), Func2() and Func3(). The psect directive has as its final entry a pointer to what is considered the first piece of executable code, the index table. (Of course, the index table is not executable, but the kernel does not attempt to execute code in a subroutine module. As long as clients do not try to execute it, all will be fine.) The table is placed at the location specified by the psect and consists of integer (long) declarations, one for each entry. There is no need to terminate the table with any special values; rather, the program using the module should be aware of the number of entries in advance. (If a program cannot be written to expect a certain number of entries, how can it possibly know what function each entry performs?

Listing 20. sub_start.a

```
*** sub_start.a ***
   nam subroutine_index        * Name of this file
   ttl simple_index            * title of this file
```

```
   use <oskdefs.d>              * an assembly '#include'

Type   equ    (Sbrtn<<8)+Obj    * equate for mod type/lang
Revs   equ    ReEnt<<8          * equate for attr/revs
Edition equ   1                 * equate for edition

   psect substart,Type,Revs,Edition,0,IndexStart

IndexStart:
   dc.l Func1
   dc.l Func2
   dc.l Func3

   ends
```

13.1.3 The Functions of a Simple Subroutine Module

We now have an index table which will point to three functions when assembled, so let's now define the functions to be pointed to. Listing 21 contains three simple functions: Func1(), Func2() and Func3(). Notice that the names of these functions exactly match the names specified in the index table code. If this is not the case, the linker will fail to find all references and thus will not complete building the subroutine module.

Listing 21. Subroutines.c

```
/*** subroutines.c ***/
/* This file contains three simple functions to */
/* demonstrate one method of implementing a */
/* shared library in the OS-9 operating system */

/* Note: If any of the routines inside a */
/* subroutine module contain references */
/* to global variables, the linker will */
/* fail to properly build the subroutine module! */
/* This is because variables must be found at */
/* link-time, not run-time. It is the nature */
/* of subroutines that the code will be found */
/* at run-time only. A subroutine module may, */
/* however, modify the contents of globals of */
/* the calling program, provided that each */
/* routine to do that receives a pointer to the */
/* variable as one of the input parameters. */

#include <stdio.h>
```

```
int Func1()
{
   /* simple function to merely return a value*/
   return(12345);
}

int Func2(int a, int b)
{
   /* this function takes two input variables and */
   /* returns their combined sum */
   return(a+b);
}

void Func3(int *a, int *b)
{
   /* here, we take two pointers to integers and */
   /* swap what each points to, a function that */
   /* could be used in a sorting routine. */
   int tmp;

   tmp = *a;
   *a = *b;
   *b = tmp;
}
```

13.1.4 The Program that Accesses a Subroutine Module

Accessing a subroutine module is a fairly trivial operation, but does require some initialization code. The client program (that is the calling program acting as a client to the server of the subroutine module) will need to link to the subroutine module (or, perhaps load it into memory) and read the index table, gathering the proper offsets before the functions can be called. This initialization is performed in Listing 22 along with the code that accesses the functions listed above.

The code presented shows a variable type you may not have seen before, a pointer to a function. Just as all "normal" variables have pointers, so do functions. The variables f1, f2 and f3 are all function pointers that will begin the program in an uninitialized state. Once the initialization function completes, these global variables will be pointing to the appropriate functions of the subroutine module and may

then be called as if the routines were located inside the local program.

Listing 22. callsub.c

```
/*** callsub.c ***/
/* This program links to a subroutine module */
/* and makes use of the functions found within. */
#include <stdio.h>
#include <errno.h>
#include <module.h>

/* define the name of the subroutine module */
#define SUB_NAME "sub_mod_test"

/* declare global variables */
int (*f1)(); /* pointer to first function */
int (*f2)();
int (*f3)();

mh_com *modptr;

main()
{
   int x, y;

   printf("Linking to subroutine module...\n");
   if (initialize()!=0){
      fprintf(stderr,"Error in initialize().\n");
      _os_exit(errno);
   }

   printf("Linked to module at address 0x%x\n",
      modptr);

   printf("Func1() returns %d\n",f1());
   printf("Func2(5,10) returns %d\n",f2(5,10));
   x = 2; y = 3;
   printf("Before swap: (x,y)=(%d,%d)\n",x,y);
   f3(&x,&y);
   printf("After swap: (x,y)=(%d,%d)\n",x,y);

   printf("Finished with demo. exiting...\n");
   deinitialize();
   _os_exit(0);
}

initialize()
{
   u_int16 attrev, tylang;
   char *mod_name=SUB_NAME;
   char path[80];
```

```
    int *funcptr;
    /* set access mode to read/exec all around */
    /* this may not need to be so verbose to work */
    u_int32 perm =    MP_OWNER_READ | MP_OWNER_EXEC | \
                      MP_GROUP_READ | MP_GROUP_EXEC | \
                      MP_WORLD_READ | MP_WORLD_EXEC;


    errno = _os_link(&mod_name, &modptr,
       (void**) &funcptr, &tylang, &attrev);

    if (errno==E_MNF) /* module not found */ {
       errno = _os_loadp(mod_name, perm,
                                          path, &modptr);
       funcptr = (int*)((int)modptr +
                        ((mod_exec*)modptr)->_mexec);
    }

    f1 = (int (*)())((int)funcptr[0]+(int)modptr);
    f2 = (int (*)())((int)funcptr[1]+(int)modptr);
    f3 = (int (*)())((int)funcptr[2]+(int)modptr);

}

deinitialize()
{
   _os_unlink(modptr);
}
```

To build the subroutine module presented, type the following commands:

```
cc -eas subroutines.c
cc -eas sub_start.a
l68 sub_start.r subroutines.rl=clib.l
-l=sys.l -o=sub_mod_test

cc callsub.c
```

and, to run the test:

```
callsub
```

13.2 Trap Modules

A trap module is another form of shared library supported by OS-9, with some considerable differences from subroutine modules. For example, though trap modules may not

access the calling program's global variables any better than a subroutine module can, trap modules are given their own static storage area by the system. (This static storage area is kept separate from that of all other uses of the same trap module; just as many processes may use one program module and each has its own static storage area.) Other differences:

1. A process may link to any number of subroutine modules simultaneously (limited by system memory) but may only link to a maximum of 15 trap modules at a time.
2. Subroutine modules may have any number of entry points provided that each is pointed to by the index table. Trap modules have exactly three entry points, only one of which is directly called actively by the calling process.
3. Subroutine module code is executed as if the routine were part of the calling process, i.e., just a simple subroutine call. Trap module code is executed by causing a hardware exception to occur through the assembly "trap" instruction.
4. When a subroutine executes, the code runs in whatever CPU state the caller is in at the time of the call. Thus, the state bit in the attributes section of a subroutine module has little meaning. When a trap module executes, the code is run in the state specified by the attributes byte of the module without regard to the state of the calling process.

13.2.1 Coding the Trap Module

Listing 23. An Assembly Trap Module

```
            nam   traphandler.a
            ttl   Example trap module, self contained

            use   /h0/defs/oskdefs.d

Edition     equ   1
Type        equ   (TrapLib<<8)+Objct
Revs        equ   (ReEnt<<8)

            psect    traphand,Type,Revs,Edition,0,TrapEnt

* if we had any static storage it would go here:
*           vsect
*VarName      ds.l  1  declare one long word
*           ends

* build the index table:
```

```
         dc.l  TrapInit
         dc.l  TrapTerm

* declare some items to be used later
String0  dc.b     "I hope you are enjoying",C$CR,C$LF,0
String1  dc.b     "this book! --Mark",C$CR,C$LF,0

*****************************
* TrapInit: gets called upon 'tlink'ing to the
* trap handler. registers passed:
*     d0.w  user trap vector number
*     d1.l  (optional) additional static storage
*     d2 thru d7 callers registers in place
*     (a0) trap module name
*     (a1) TrapEnt entry point
*     (a2)  this module
*     a3 thru a5 callers registers in place
*     (a6) static storage for trap handler
*
* returns: carry flag set on error, with error
*     code in d1.w.
*
* Stack image upon entry to TrapInit:
*
*        [ return address ] <4 bytes>
*        [    00000000    ] <4 bytes>
*  sp -> [ user's  a6 reg ] <4 bytes>
*
* before exiting, TrapInit must move the sp so
* it points to the return address

TrapInit:
* First, do any trap specific items, such as
* initializing static storage. This routine
* does not have anything to do, so clean up
* the stack and exit.
         movem.l (a7),a6      * restore the user's a6
         addq.l   #8,a7       * restore stack ptr
         rts                  * return to caller

*****************************
* TrapEnt: gets called thru 'tcall'ing to the
* trap handler. registers passed:
*     d0-d7 callers registers in place
*     a0-a5 callers registers in place
*     (a6) trap handler's static storage
*
* Stack image upon entry to TrapEnt:
*
*        [ return addr ] <4 bytes>
*        [ vector   #  ] <2 bytes>
*        [ function #  ] <2 bytes>
*  sp -> [ user's  a6  ] <4 bytes>
*
```

```
* before exiting, TrapEnt must move the sp so
* it points to the return address
*
         org   0
* after saving items of the stack, these labels will
* be offsets to each item on the stack
S.d0     do.l  1
S.d1     do.l  1
S.a0     do.l  1
S.a6     do.l  1
S.func   do.w  1
S.vect   do.w  1
S.pc     do.w  1

TrapEnt:
         movem.l  d0-d1/a0,-(a7)     * save registers
         move.w   S.func(a7),d0      * get func code
         cmp.w    #1,d0    * compare it to (u_int16)1
         bhi.s    TooHigh            * if higher, branch
         beq.s    Func1          * func = 1, execute it
*implied: bne.s   Func0 * execute function 0
Func0
         lea      String0(pc),a0     * get first string
         bra.s    OutPutIt           * and display it

Func1
         lea      String1(pc),a0     * get second string
*implied: bra.s   OutPutIt           * and display it

OutPutIt
         moveq    #1,d0        * place stdout path in d0
         moveq    #80,d1       * max 80 chars for writeln
         os9      I$WriteLn    * and call writeln()
         bcs.s    ErrReturn    * branch if write failed
CleanAndReturn
         movem.l (a7)+,d0-d1/a0/a6     * pop regs from stk
         addq.l   #4,sp        * and skip vect/func
         rts                   * return to caller
TooHigh
         move.w   #(12<<8)+12,d1     *error 012:012 in d1
ErrReturn
         move.w   d1,D.d1+2(a7)      * give user err code
         ori      #Carry,ccr         * and tell kernel
         bra.s    ClearAndReturn
*****
* In theory, this routine should never
* be called. In case it does, since I don't
* know what to do, I will cause the calling
* process to exit with error 012:012
TrapTerm
         move.w   #(12<<8)+12,d1
         os9      F$Exit

         ends
```

13.2.2 The Program that Accesses a Trap Module

Listing 24. A Trap Calling Program

```
            nam       trapcaller.a
            ttl       Trap testing program

            use       /h0/defs/oskdefs.d

Edition     equ       1
Type        equ       (Prgrm<<8)+Objct
Revs        equ       (ReEnt<<8)

            psect     traptst,Type,Revs,Edition,512,Main

Vector      equ       5
Name        dc.b      "traphandler",0

Main
            moveq .l  #Vector,d0      * trap vector in d0
            moveq.l   #0,d1           * no additional memory
            lea       Name(pc),a0     * trap name in a0
            os9       F$TLink         * link to the handler
            bcs.s     Exit            * exit if error

            tcall     Vector,0        * call function 0
            bcs.s     Exit

            tcall     Vector,1        * call function 1
            bcs.s     Exit
            moveq.l   #0,d1        * clear the error register

Exit
            os9       F$Exit

            ends
```

To assemble a trap module and test program, type the commands:

```
cc -eas traphandler.a
l68 traphandler.r -l=sys.l -o=traphandler
cc -eas traptest.a
l68 traptest.r -l=sys.l -o=traptest
```

and, to execute the test program:

```
traptest
```

CHAPTER 14

Exception Handling through C

An exception, in the context of this chapter, is what happens when software performs some (generally) unintentional action to upset or confuse the CPU.

Normally, when a program takes a bad turn ending up in a hardware exception (such as division by zero or a bus error), the kernel automatically terminates the offending program as a means of dealing with this condition. However, if the program had previously informed the kernel concerning the address of an exception handling function, the kernel instead forces the offending process to execute that function, asynchronously.

14.1 Types of Exceptions

There are several hardware exceptions that can be generated by the 68XXX hardware. The standard include file <sysglobs.a> has symbolic names and values for all of the

exceptions supported under OS-9. The names and definitions of the most common of these are seen in Table 7.

TABLE 7 Hardware Exception Types

Mnemonic	Meaning
T_BUSERR	Bus error. An attempt to access non-existing or privileged memory.
T_ADDERR	Address error. An attempt to access memory on an improper boundary.
T_ILLINS	Illegal instruction. The machine code is improper.
T_ZERDIV	Division by zero.
T_CHK	CHK instruction. This instruction is designed to cause an exception.
T_TRAPV	TRAPV instruction. This instruction is designed to cause an exception.
T_PRIV	Privilege violation. Some instructions must execute in supervisor state.
T_1010	A-Line exception. Reserved for machine codes beginning with a hex A.
T_1111	F-Line exception. Reserved for machine codes beginning with a hex F.

Any of the above exceptions could occur if a program has not been written with enough care. Although most of the exceptions could easily be looked for before they occur, this amount of overhead can bring a formerly fast program to its knees. There are also times when it is actually desirable to cause an exception, such as the bus error if a program is probing for memory existence or accessibility. (An MMU can prevent processes from accessing existing memory.)

If a program causes an exception, the kernel will generally terminate the offending process, but this is not always the best course. If a program is written with the intentions of perhaps causing an exception, it can also include code which will allow the process to continue running after the

exception occurs. This involves telling the kernel the addresses of routines to run after an exception occurs.

14.2 Telling the System About Your Exception Handler

The _os_strap() system call binding is used to tell OS-9 about your exception handling routines, including where they are and what exceptions they handle. Using this call requires including the header file <settrap.h> to access a structure definition used in setting up pointers to your exception handlers. Listing 25 contains a simple program which will quickly generate a bus error and allow the system to abort it. (That is, it passively allows the system to terminate the process because no exception handlers are installed in the program.)

Listing 25. berr1.c

```
/*****
** berr1.c: this program generates a bus error and
** is terminated afterwards.
**
*****/
#include <stdio.h>
#include <MACHINE/reg.h>
main()
{
   int *address = (int*)0xfffffffc;

   printf("Generating a bus error...\n");
   *address = 1;
   /* the previous line causes a bus error unless */
   /* your hardware uses address 0xfffffffc as */
   /* valid memory access */

   printf("If the program gets here, something went
            wrong\n");
}
```

In Listing 26 the program is taking control of its bus error exception. There is a global variable of type strap which is used to hold information for each exception handler we wish to install. There is not a single routine to handle excep-

tions for a process, but a routine per exception per process. However, most programs will not install routines for all exceptions and occasionally programs will use one handler routine for multiple exception types. These must be installed implicitly, however.

The variable called error_table is the strap type which defines the exceptions the program will handle and how. This variable is an array of structures which *must* be terminated so the _os_strap() function knows to stop looking at data. Each structure in the array contains two values, an exception vector symbol and the name of the routine serving as a handler for the exception. The terminating element contains -1 as the vector value.

The _os_strap() call takes two parameters, a pointer to a stack to use for all exception handling routines and a pointer to the strap table. If the stack pointer is non-zero, when any exception handler for this process is called, that pointer will point to the stack the routine uses. If instead the pointer is zero, the routine will use whatever stack pointer is currently in place. If an exception handling routine may use a large amount of stack space, it is a good idea to allocate a special stack prior to the _os_strap() call, and pass in a pointer to the highest address in this special space. Note that all exception routines for a single process will use the same stack pointer. (Of course, a routine could allocate its own stack on the fly.)

Once a routine is installed as an exception handler, if you would like to stop handling the particular exception, the _os_strap() call may be used again, this time with 0 as the corresponding routine in the strap structure for the particular exception. If this action is taken and the process subsequently causes the exception to occur, the kernel will terminate the offending process.

Listing 26. berr2.c

```
/*****
** berr2.c: this version of the program installs a
** bus error handler to be called when that exception
** is generated.
**
*****/
#include <stdio.h>
#include <types.h>
#include <errno.h>
#include <settrap.h>
#include <MACHINE/reg.h>

#define DEFAULT_STACK 0

/* Remember, Ultra C is an ANSI compiler */
u_int32 BerrFunc(u_int32, u_int32, u_int32);

strap error_table[] = {
   { T_BUSERR, BerrFunc },
   { -1, NULL }
};

main()
{
   int *address = (int*)0xfffffffc;
   if (errno=_os_strap(DEFAULT_STACK,
    &error_table[0])){
      fprintf(stderr,"Error %d installing exception
                          handler(s)\n",errno);
      _os_exit(errno);
   }

   printf("Generating a bus error...\n");
   *address = 1;

   printf("If the program gets here, something went
         wrong\n");
}

u_int32 BerrFunc(u_int32 vector, u_int32 badpc,
                     u_int32 badaddr)
{
   fprintf(stderr,"A bus-error occured\n");
   fprintf(stderr,"The error occured at code address
                        0x%x\n",badpc);
   fprintf(stderr,"The offending address is 0x%x\n",
                        badaddr);
   _os_exit(vector);
}
```

Notice that the exception handling routine in the last example terminated the program with an _os_exit() call. If, rather than exit, the routine was allowed to simply return, the program would either generate another exception, get caught in an exception loop, or terminate on its own anyway. (Different versions of the C compiler took different actions.) This is because when the 68k processor reaches an rts assembly instruction, the chip expects the return address to be pointed to by the stack pointer. When the kernel causes an exception routine to run, there is no subroutine instruction call to bring the process there, the kernel simply changes the process' program counter register, thus no return information is placed on the stack. (The branch-to-subroutine instructions [and jsr] of the 68k command set all push the address of the instruction following the subroutine call onto the stack so that the corresponding return-from-subroutine [rts] call will work. For the rts to work, the stack pointer must be pointing to the return address.) It is possible to avoid terminating the program inside an exception handler and instead get back into the main program. This requires a sort of a goto instruction — a long jump. (If you are not familiar with the goto instruction, do not feel left out. If you are familiar with it, I'm sorry.)

14.3 Handling the Error Without Terminating Your Program

There is a function, longjmp(), which allows your program to start executing at a location that has been previously arranged. The location longjmp() jumps to is set up through the setjmp() call. setjmp() is called prior to the longjmp() function, and fills a structure with enough information to allow longjmp() to reinstate the register set of the process at the time of the setjmp() call.

The setjmp() function will return 0 the first time it is executed. Each time the call is returned to through a longjmp() call, the return value is at the discretion of longjmp() (or,

more appropriately, the software engineer). These functions require including the standard header file <setjmp.h> which defines a structure of type jmp_buf. A variable of this structure type is used by setjmp() to save the current register image of the process at the initial call and by longjmp() to restore the image set. In addition, longjmp() takes a second parameter, the value setjmp() should return as a result of being reached through longjmp(). The use of these two functions is shown in Listing 26.

Listing 27. berr3.c

```
/*****
** berr3.c: this version of the program installs a
** bus error handler to be called when that exception
** is generated,
** and returns back to the main program.
**
*****/
#include <stdio.h>
#include <types.h>
#include <errno.h>
#include <settrap.h>
#include <setjmp.h>
#include <MACHINE/reg.h>

#define DEFAULT_STACK 0

/* this global will hold the register set */
jmp_buf JumpBuffer;

u_int32 BerrFunc(u_int32, u_int32, u_int32);

strap error_table[] = {
   { T_BUSERR, BerrFunc },
   { -1, NULL }
};

main()
{
   int *address = 0xfffffffc;
   int return_code;

   if (errno=_os_strap(DEFAULT_STACK,
                                    &error_table[0])){
      fprintf(stderr,"Error %d installing exception
                          handler(s)\n",errno);
      _os_exit(errno);
   }
```

```
    printf("Generating a bus error...\n");
    return_code = setjmp(JumpBuffer);
    if (return_code == 0) {
       /* first time through this code */
       *address = 1;
    } else {
       /* not the first time through! */
       printf("It seems that an exception has
                occured!\n");
       printf("The longjmp() code is %d\n",
                return_code);

       /* this exit call is entirely optional. */
       _os_exit(0);
    }

    printf("If the program gets here, something went
             wrong\n");
}

u_int32 BerrFunc(u_int32 vector, u_int32 badpc, u_int32
badaddr)
{
    fprintf(stderr,"A bus-error occured\n");
    fprintf(stderr,"The error occured at code address
                          0x%x\n",badpc);
    fprintf(stderr,"The offending address is 0x%x\n",
                          badaddr);
    /* WAS: _os_exit(vector); */

    /* 102: the bus-error error code */
    longjmp(JumpBuffer, vector);

    printf("Code never gets here!\n");
}
```

CHAPTER 15

Interrupt Service Routines

Hardware external to the CPU communicates with the CPU and any programs it is running by sending interrupts. OS-9 has a very specialized way of handling hardware interrupts which is described in this chapter.

15.1 Interrupts vs. Exceptions

The word exception as used in this chapter takes a slightly different meaning than in the last chapter. In Chapter 14, the word refers to some specific hardware error conditions. In this chapter, the word is more generic, taking into account all of the possible vector hits, whether related to error conditions or not.

Interrupt can mean the same thing as exception taken in the generic sense, but this chapter takes the word interrupt to mean some specific vectors, the vectors that are reserved for I/O purposes.

15.2 The Standard 68k Interrupt Mechanism

The 68k family of processors maintains a 256 element index, called a vector table, which is consulted when one of the 256 possible exceptions occur. This table is built containing a 32 bit absolute address at each entry point; that address is the location in memory containing code to execute when its particular exception occurs. Thus, if exception 2 occurs, the CPU looks to see where element 2 of the vector table points, and begins executing code from that point.

➤ The offset into the vector table for a particular exception is always the vector number times four, since each entry is a 32 bit integer (four bytes).

15.3 OS-9's Unique Approach to Interrupt Handlers

Under OS-9, the system takes a great deal of control concerning all of the hardware exceptions and the vector table. This is because, if the operating system is to continue running after an exception occurs, the handling routine must conform to some very rigid specifications. Rather than force the software engineer to learn these rules, the system provides hooks to install routines which the kernel will call when the appropriate exception occurs.

OS-9 fills the vector table with pointers that bring the CPU into the system kernel itself. Installing an interrupt on the OS-9 system does not involve modifying the vector table, instead it involves modifying OS-9's interrupt polling table.

There can be several interrupt routines installed on a single vector by placing pointers to each routine along the polling list for the particular vector. When this vector is interrupted, the system will call the first interrupt service routine in the vector's list. This routine has the responsibility of telling the kernel whether or not the interrupt was generated by the hardware it services. If its hardware caused the interrupt,

the routine should service the hardware and then tell the kernel. If not, it should return to the kernel quickly so the kernel can move to the next interrupt service routine in that vector list. (When a routine tells the kernel it *did* service the interrupt, the kernel will stop calling routines in that vector list.)

The order in which routines installed on a vector are called depends on a priority assigned to each routine. Interrupt priorities (which have nothing to do with process priorities) range between zero and 255. One (1) is the highest value an interrupt priority can hold. (Zero is special and will be discussed shortly.) When an interrupt routine is installed on the vector, a priority is one of the parameters to the system call. Based on this priority, the kernel sorts the routine into the list similarly to how the kernel sorts a process into the active queue. Starting from the rear of the list, as the priority of the routine being inserted is greater than the routine being compared to, the new routine keeps moving forward. When the comparison has each priority equal or the new routine lower (higher number), the routine being installed is placed behind the routine it was just compared to.

If a certain piece of hardware's interrupt service routine cannot read it to determine if it generated the interrupt, this hardware will have a difficult time fitting into the interrupt model described above. (If the software can't determine the status of its hardware, how does it know how to respond to the kernel's question, "Did your hardware generate this interrupt?") To compensate for this situation, the service routine should be installed with the special priority of zero. This priority demands that the routine be the *only* routine installed on the vector. If, at the time of installation, there is already another routine on the vector, the new installation request will fail. By the same token, if a routine with priority zero is on a vector and another routine attempts to install itself on that vector at *any* priority, the attempt at a new routine installation will fail.

➤ Priority zero is also used for interrupts that are expected to be generated often, such as clock tickers. Since these interrupts occur very frequently, it would be a waste of time for the associated interrupt service routine to check for the interrupt-pending condition of the hardware with each interrupt. Since the routine is installed with priority zero, there should be no other sources for the same interrupt vector and the ISR can safely assume that its hardware generated the interrupt. With the addition of "Fast IRQs" to OS-9 3.0, priority zero interrupts may no longer be necessary on the polled table.

15.3.1 Finding Out What Interrupts are Currently Installed

The irqs command shows the interrupt service routines currently installed on the system. Also shown with this command is each ISR's vector number, interrupt priority, hardware (port) address, static storage, driver name and, if available, device (descriptor) name.

➤ It is possible for code located within a module other than a device driver to install an interrupt service routine. The name listed as the driver is the name of the module containing the service routine.

FIGURE 22 Irqs Command Output

```
$ irqs
Vite_MVME167  OS-9/68040 V3.0  (max devs: 128, max irqs: 64)

 vector   prior   port addr   data addr    irq svc     driver   device
-------   ------  ---------   ----------   ---------   -------  ------
101       5       $fff47000   $00ffc410    $00020d6a   scsi167
103       0       $fff46000   $00eaea30    $00db2c94   if167    ie0 (?)
104       5       $fff47000   $00ffc410    $0002278e   scsi167
105       1       $fff40000   $00000000    $0001c00e   tk167
112       5       $fff45000   $00ffd420    $0001de44   sc167    term
120       5       $fff45000   $00ffa360    $0001de44   sc167    t1
123       5       $fff45000   $00ffa360    $0001deae   sc167    t2
124       5       $fff45000   $00ffa2c0    $0001de44   sc167    t3
$
```

15.3.2 Installing a System Polled Interrupt Service Routine

The call _os9_irq() is used to install a new ISR on the systems standard interrupt polling table. The parameters passed into this call are the interrupt's vector number, the polling priority, a pointer to the service routine, a pointer to be used as the static storage area for the ISR and a pointer considered to be the hardware address of the device serviced.

The hardware address is not used directly by the kernel since the kernel has no idea of the device's register format. However, the ISR gets this hardware address in a register whenever the ISR is called by the system interrupt poller. This, combined with getting the static storage address in another register, allows the same interrupt service routine, in the same device driver or application, to be used in conjunction with several separate devices. (Assuming, of course, that none is installed with priority zero.)

15.3.3 Installing a "Fast" Interrupt Service Routine

A similar call for installing ISR's, _os_firq() takes the same parameters as _os9_irq() excluding the hardware pointer, which is not needed. Since only one ISR may be installed on a fast vector this call is roughly equivalent to installing a polled ISR at priority zero. However, the fast irq has an additional benefit in that the routine will be reached slightly sooner, since only registers d0 and a2 are saved for the routine vs. all registers in the polled ISR. Also, a fast ISR does not have the support of the system for hardware exceptions while a normal ISR does.

15.3.4 Removing an Interrupt Service Routine from the System

For both standard and fast interrupts, the removal of the ISR involves calling the same routine used to install the ISR with all parameters exactly as used previously with the

exception of the address of the service routine; that parameter must be zero.

15.4 A Simple Program to Install an Interrupt Routine

The conclusion of this chapter presents a simple case for installing an intercept routine. The program's source in Listing 28 installs an interrupt service routine that will do nothing more than increment a counter located inside a data module. Since this routine does not have a piece of hardware to support or generate interrupts, we have a small problem. (Actually, lack of hardware to support is no problem, but lack of interrupt generation is certainly a problem.)

The approach this program takes is to install an interrupt on vector 255 at priority 255. For testing it, you must have some source of interrupts and you may need to modify the code. Modifications you make should install the routine on a vector that is already in use on your system, at a priority that is higher (lower in number) than the routine whose interrupt you are borrowing. (Using this approach should force your handler to tell the kernel it has *not* handled the interrupt, so it may be passed to the proper interrupt handler.)

An option to changing the code is to somehow cause the routine to execute even though a hardware interrupt has not really occurred. On the optional companion disk to this book is a "magic" program which causes an interrupt routine to be called without hardware present. This program has been written to be generic enough to cause any desired interrupt service routine to be called, allowing the utility to be used in testing drivers before hardware is available. Of course, if the routine tries to access non-existing hardware, unpredictable results will follow.

In Listing 28, notice toward the top of the program that there are many #define's detailing certain aspects of how

the program will run, including the vector and priority of the service routine. Also, notice that there is some assembly language inside this program. This cannot be avoided since register conventions between ISR's and normal C code are not the same. One more thing to point out is that this code must run in system state, since the routine to install an ISR must be executed from system state.

➤ The assembly language for this program is entirely contained inside the C file. For information on how this is accomplished, see Chapter 12.

Listing 28. Simple Interrupt Service Routine Installation

```
/*** irq.c ***/

/* if you modify this program to make it use */
/* an existing interrupt that another irq */
/* is supposed to handle, remove the next line */
#define NOT_MY_IRQ

/* and, these defines also must be changed if */
/* your routine has its own hardware too */
#define VECTOR 255
#define PRIORITY 255
#define PORT_ADDR 0x00112233

#define DM_NAME "IRQ_DATMOD"

#include <stdio.h>
#include <module.h>
#include <procid.h>
#include <setsys.h>
#include <types.h>
#include <errno.h>

u_int32 *irqcount;

/* make sure this program is run in system state */
/* by turning on that bit in the attributes */
/* ATTR: 0xA0, REVS: 0x00 */
_asm("_sysattr: equ $a000");

void *_ISR();

main()
{
   procid *prc = (procid *)sysglob(procid*, D_Proc);
   u_int16 attr, type;
   u_int32 perm, dm_size;
```

```
mh_com *mod_head;
u_int32 zero=0;

/* fix up stdio globals to work properly */
/* from system state */
stdin->_fd = prc->_path[0]; prc->_path[0] = 0;
stdout->_fd = prc->_path[1]; prc->_path[1] = 0;
stderr->_fd = prc->_path[2]; prc->_path[2] = 0;
_from_new(stdin);
_from_new(stdout);
_from_new(stderr);


/* create the data module */
dm_size = sizeof(*irqcount);
type = (MT_DATA<<8);
attr = (MA_REENT<<8);
perm = MP_OWNER_READ | MP_OWNER_WRITE;

errno = _os_datmod(DM_NAME, dm_size, &attr,
   &type, perm, (void**)&irqcount,
   (mh_data**)&mod_head);

if (errno == E_KWNMOD) { /* already exists */
   fprintf(stderr,"Error, %s already exists!\n",
      DM_NAME);
   _os_exit(errno);
}

if (errno != 0) { /* some other error */
   fprintf(stderr,"Fatal error creating data
      module!\n");
   _os_exit(errno);
}

/* initialize the counter */
*irqcount = 0;

/* install the service routine */
errno = _os9_irq(VECTOR, PRIORITY, _ISR, a6(),
   PORT_ADDR);
if (errno!=0){
   fprintf(stderr,"Error installing interrupt\n");
   _os_exit(errno);
}

/* while (!condemned) sleep(0) */
while ((prc->_state&PS_CONDEMN)==0)
   _os9_sleep(&zero);

/* remove ISR */
_os9_irq(VECTOR, PRIORITY, NULL, a6());

/* unlink from DM and exit */
```

```
    _os_unlink(mod_head);
    _os_exit(0);

}

_asm("_ISR:");
/* first, fix register(s) */
/* then call the real routine */
_asm("   move.l a6,-(a7)");
_asm("   move.l a6,d0"); /* sysglobs parameter */
_asm("   move.l a3,d1"); /* portaddr parameter */
_asm("   movea.l a2,a6");/* globals ptr */
_asm("   bsr ISR");
/* now, check return value and set/clear CCR */
_asm("   move.l d0,d1");
_asm("   beq HandledIt");
/* not my interrupt, set carry flag */
_asm("   ori.b #01,ccr");
/* restore registers and exit */
_asm("HandledIt");
_asm("   move.l (a7)+,a6");
_asm("   rts");
_asm("a6:");
_asm("   move.l a6,d0");
_asm("   rts");



ISR(u_int32 *sysglobs, u_int32*portaddr)
{
#ifdef NOT_MY_IRQ /* non-modified program */
    (*irqcount)++;
    return(0); /* 0: I handled interrupt */
#else
    /* do whatever your hardware needs! */
    /* return(0 or 1);*/
#endif
}
```

Once this program is placed onto your system, it may be compiled with a simple:

cc irq

and should run in the background with a simple:

irq&

but will only work if you compile and run as a super user! This is because the program is a system state program. OS-9

maintains this security in allowing system state programs to be used, since a program running in system state may access any memory area on the system.

The program should run in the background so you will still have access to your terminal. Now, you need a method of verifying that the interrupt service routine is doing anything. The program shown in Listing 29 links to the data module used, reads the current value of the counter being incremented with each interrupt, reports this value and unlinks from the data module prior to exiting.

Listing 29. Verifying the Routine Works

```
/*** irqcount.c ***/
#define DM_NAME "IRQ_DATMOD"

#include <stdio.h>
#include <module.h>
#include <types.h>
#include <errno.h>

u_int32 *irqcount;

main()
{
   u_int16 attr, type;
   mh_com *mod_head;
   char *dm_name = DM_NAME;

   /* link to data module */
   type = (MT_DATA<<8);
   errno = _os_link(&dm_name, &mod_head,
      (void**)&irqcount, &type, &attr);

   if (errno==E_MNF){
      fprintf(stderr,"Error: module not found!\n");
      _os_exit(errno);
   }

   if (errno != 0){
      fprintf(stderr,"Fatal error linking to data
         module!\n");
      _os_exit(errno);
   }

   /* get value and display it */
   printf("The number of interrupts so far: %d\n",
      *irqcount);
```

```
    /* unlink and exit */
    _os_unlink(mod_head);
    _os_exit(0);
}
```

Once this code is compiled you can see how many times the interrupt service routine has been called. But, do you have an interrupt source? If you've modified your code as noted above to use interrupts that are generated by other hardware on your system, you can view a changing counter. If not, if you have the interrupt utility from the companion disk, you can cause the system to call your routine with the command:

```
interrupt -v=255 -p=255
```

which causes the interrupt service routine on vector 255 and at priority 255 to be executed. (If there is more than one routine on this vector and at this priority, the utility will call the first one found. The irqs utility will show you how many routines are installed and in what order.)

➤ The interrupt utility will, by default, cause the routine to be executed only once. Multiple irq's can be simulated with the optional flag "-c=<count>" where <count> is an integer telling the number of times to call the routine.

15.4.1 Masking Interrupts

Although it is not necessary for the above program, interrupt service routines must occasionally mask the CPU from receiving other interrupts to prevent the ISR from being interrupted itself. This small routine, written in C-embedded 68k assembly language, will mask interrupts to the hardware level passed in as the only parameter. This routine will also return the level of masking the CPU had prior to this change, allowing the routine to restore the masking level when it is ready.

Listing 30.

Listing 31. Masking Interrupts

```
/* prototype: */
/* u_int16 mask_irq(u_int16); */
/* use: */
/*    oldlevel = mask_irq(newlevel); */
_asm("mask_irq:");
_asm("   clr.l -(sp)"); /* keep stack lw aligned */
_asm("   move.w sr,2(sp)"); /* save current sr *
_asm("   move.w   sr,d1");
_asm("   or.w     d0,d1"); /* or in the new bits */
_asm("   move.w   d1,sr"); /* set new status reg */
_asm("   move.l   (sp)+,d0"); /* return old sr */
_asm("   rts");

/* prototype: */
/* void restore_irq(u_int16); */
/* use: */
/*    restore_irq(oldlevel); */
_asm("restore_irq:");
_asm("   move.w   d0,sr");
_asm("   rts");
```

The above code requires you to call mask_irq() with the value to or into the status register. If you want to mask interrupts to a particular level, you must shift a short (representing the mask value) eight bits to the left and use that as your parameter to mask_irq(). For example, to mask interrupts to level seven (all but NMI's masked), you would place a call as `oldlevel = mask_irq(7<<8)`.

CHAPTER 16

OS-9 System Global Variables

The OS-9 operating system, like any regular program, maintains a number of global variables. To make the system as flexible as possible, the kernel's globals are readable by any process running and super user processes may alter most of the globals as necessary. This chapter discusses several of the variables and why they might be altered while the system is running.

16.1 More on Process Scheduling

16.1.1 Altering the Effects of Aging

Although the designers of the OS-9 operating system thought the aging mechanism was a good idea, there are some users of OS-9 who may prefer to design a strict priority-based system. To allow for this, OS-9 has a global which decides how far a process may age as a maximum. This variable, called d_maxage, is usually set to zero, meaning process scheduling works as described in Chapter 11. If d_maxage is non-zero, no process may age beyond (or up to) its value. If you wish to remove all aging from the system, simply set d_maxage to 1.

➤ If d_maxage is being used, a process entering the active queue is given an age equal to its priority even if its priority exceeds d_maxage. However, in this case the process' age will not increase.

16.1.1.1 Viewing the Contents of a System Global

A system global's contents may be copied into one of your local variables with the _os_getsys() system subroutine. This call requires three parameters: an offset into the system global area for the variable you are interested in, the size (in bytes) of the global and a pointer telling the compiler where to place the copy of the global.

The standard include file <sysglob.h> contains a structure representation of the system global area in the typedef sysglobs. Since this structure accurately portrays the storage of system globals, it can be used to obtain the size and offset for any global you need. The offset of any global you need may be received through the offsetof macro which is defined in the <stddef.h> header file. See Listing 32 for an example of _os_getsys() using the offsetof macro.

Listing 32. seemaxage.c

```
/*** seemaxage.c ***/
#include <stdio.h>
#include <stddef.h>     /* bring in offsetof macro */
#include <sysglob.h>    /* get structure elements */
#include <types.h>      /* get extended type defs */
#include <errno.h>      /* get errno global */
main()
{
   u_int16 d_maxage;
   errno = _os_getsys(offsetof(sysglobs, d_maxage),
      sizeof(u_int16), (glob_buff*)&d_maxage);
   if (errno!=0)
   {
      fprintf(stderr,"Error getting the global!\n");
      fprintf(stderr,"Contact your book author!\n");
      _os_exit((status_code)errno);
   }
   else
   {
      printf("d_maxage is %d\n",d_maxage);
   }
}
```

16.1.1.2 Changing the Contents of a System Global

If your process is running as a super user (that is, a member of group zero), it may modify certain global variables. The routine called _os_setsys() is used to change variables. Its parameters are basically the same as those for _os_getsys() except the last, which is not a pointer to a storage location, rather the value to which you would like the global you refer to changed. Listing 33 provides an example of changing a system global.

Note: Be *very* careful about changing system globals as blind experimentation can quickly crash a system!

Listing 33. setmaxage.c

```
/*** setmaxage.c ***/
#include <stdio.h>
#include <stddef.h>      /* bring in offsetof macro */
#include <sysglob.h>     /* get structure elements */
#include <types.h>       /* get extended type defs */
#include <errno.h>       /* get the errno global */
main()
{
   glob_buff d_maxage;
   printf("What should the new d_maxage be?");
   scanf("%hd",&d_maxage.wrd);

   errno = _os_setsys(offsetof(sysglobs, d_maxage),
      sizeof(u_int16), d_maxage);
   if (errno!=0)
   {
      fprintf(stderr,
         "Error setting global. Are you SUPER?\n");
      _os_exit((status_code)errno);
   }
   else
   {
      printf("d_maxage has been changed!\n");
   }
}
```

16.1.2 Suspending Processes

OS-9 does not provide a direct method of suspending a process, but through a combination of priorities and the global d_minpty, a process can be put into a simulated state of suspension.

➤ A state of suspension is simulated because the state simply does not exist on OS-9. OS-9000, however, does allow for process suspension through the _os_suspend() call.

The d_minpty variable is also usually zero, meaning it is currently unused. If this global is non-zero, the lowest priority process will be permitted to execute. Any process with a priority below d_minpty will be in this simulated state of suspension. This variable can be potentially dangerous because, if there are no processes on the system with a priority *greater* than d_minpty (equal is not good enough unless d_minpty is zero), the system will come to a quick halt.

16.1.3 Ticks Per Time Slice

Earlier it was mentioned that the default number of ticks in a process' time slice is two and that it could be changed a couple of ways. The easiest way is to simply change the global d_tslice. This system global is an integer defining the number of clock ticks each time slice will pass as a maximum. This value should not be set any lower than two or results will be unpredictable.

16.2 Measuring Time

16.2.1 Unitless Measurement

d_ticks is a variable that is set to zero upon system start-up and is incremented by one each time the clock interrupt occurs and is therefore very useful for taking time measurements, as long as the resolution of the system clock is sufficient. To time something, just record d_ticks to start timing

and record it again to complete timing. The difference between the two is the number of ticks that have occurred. One concern when taking time measurements is detecting rollover of the tick counter. Of course, it is simple to detect a single rollover; the finish timestamp will be smaller than the start timestamp. But, if this has happened, how many times might the counter have rolled over? Unfortunately, there is no way to determine the answer to that question but it may not be very likely to happen. If the system clock is set to interrupt 100 times per second (the default), the counter will roll over approximately once every year and four months. If the clock increases to 1000 ticks per second (perhaps a practical maximum), this changes to once every seven weeks.

➤ d_ticks is a variable that should not be modified, only read.

16.2.2 Gaining Meaning From this Measurement

Great, so now you know how to measure the amount of ticks something takes to happen, but just how long is that? The global d_tcksec holds the number of ticks that are sent per second and is thus useful for turning a tick count into a real time measurement, no pun intended.

It may be tempting to crank out higher resolution by changing d_tcksec during the timing loop. This action would not actually change the tick rate, however, as the variable is only present for informational purposes.

16.3 Determining Processor Types

16.3.1 CPU Type

The CPU type can be determined by reading the global called d_mputype. This variable is an integer equal to the type of CPU, such as 68030. It is set by the kernel at boot time and should not be modified.

16.3.2 FPU Type

You can determine whether or not a floating point unit (FPU) is present, and if so, what kind by reading the global d_68881. The global can currently be one of 0, 1, 2 or 40, corresponding to no FPU, 68881, 68882 and 68040 which has a built-in FPU. (Although it is too soon to tell, the 68060 CPU with built-in floater will probably be recorded as 60 in d_68881.)

16.4 Other Globals

16.4.1 System State Preemption

Prior to OS-9 Version 3.0, the rule regarding code executing in system state was that system state code could not be preempted or time-sliced unless the code did something to specifically allow a time slice to occur, such as enter the sleep queue. Version 3.0 of the system has introduced a new concept, which allows for system state code to be time sliced. There will be certain times, however, when time slicing during system state code may be undesirable, so a new system global has also been introduced to control this effect. The global called d_preempt is a 32 bit unsigned integer which, if equal to zero, dictates that system state time slicing may occur, and if non-zero, no system state time slicing may occur.

16.4.2 Pointer to the Current Process Descriptor

The global called d_proc always points to the process descriptor of the process currently in control of the CPU. When an interrupt is being serviced, this variable points to the descriptor of the process that was running at the time of the interrupt. The data in this (or other) process descriptors should not be modified or unpredictable results may follow.

There are many other system global variables that may be of interest. Most of them should not be modified as the kernel depends heavily on the proper values stored in them.

CHAPTER 17

Memory Management on the OS-9 System

Managing memory for a process can often be a tricky issue. This chapter will present the major system calls involving memory.

17.1 Memory List

For the OS-9 system to be aware of any memory available, each area of physical memory must be included in the MemList, an area inside the init module. This memory list defines all possible areas of memory (RAM, ROM and NV-RAM) that OS-9 should search for during the boot procedure. Part of the boot procedure is to parse the memory list and then validate each area included to dynamically decide what memory is present and what is not. (If a certain area of memory is part-time memory, for example, only present during development, it can be in the list full-time with no repercussions.)

Although parts of an init module (in its file on disk) may be modified with the moded utility, the memory list may not. Rather, any changes must be made in the source code and the module re-assembled. The location of the source to init may be specific to all systems, but on a resident OS-9 sys-

tem a good place to search would be the /dd/SYSMODS directory, if present. In this location you may find the file init.a. One *gotcha*, however, is that the MemList definition is often not included in this file, rather in a file called systype.d which is included by init.a. This file is often found in the system's default header-file directory.

➤ On a cross-development system there is a subdirectory area for OS-9 called MWOS. In this MWOS structure, the init.a source file should be found in MWOS/OS9/SRC/SYSMODS/INIT/init.a (whew, that's a lot to take in) and the definition file it uses will be MWOS/OS9/<CPUtype>/PORTS/<boardname>/systype.d. This last path has two variable items within it: <CPUtype> and <boardname>. <CPUtype> may be either 68000, 68020 or CPU32, with 68000 being valid for a 68010, 68020 valid for a 68020, 68030, 68040, 68060 (presumably, since that chip is not available at the time of this writing) and CPU32 valid for the 683xx series of CPUs. <boardname> is a reference to the system port specific to your hardware and the names available will vary. Currently, Microware only ships ports to CPU boards made by Motorola, other vendors supply ports to their own hardware.

Once the system has found all of the memory listed, it builds an internal list of memory so it can begin using it in a controlled manner including the allocation of memory to processes and the taking back of memory returned by processes. To view all areas of free memory the command mfree accepts a -e flag for extended information.

17.2 Methods of Memory Allocation

Beginning with OS-9 Version 3.0, there are two methods of memory allocation available. The allocator that has been with OS-9 previous to Version 3.0 is called the first-fit allocation method, meaning a memory request will be granted from the first free area of memory large enough to accommodate the request, fitting the priority and color requirements, if any, of the request. This allocation method is not a bad one since it is fairly efficient with memory. However, it is not a good one (for real-time use) since the amount of time it takes to fulfill a memory request is dependant on

some factors that make it very difficult to determine in advance.

New to OS-9 Version 3.0 is the buddy allocation method, which performs similarly (time-wise) to the standard (first-fit) allocator when both systems have an ideal degree of memory usage and fragmentation, but the buddy allocator has a worst-case time to fulfill the request that is very close to its best-case time. However, the blocks of memory the buddy allocator will grant must be a power of 2 in size, thus a request for slightly more than a half megabyte of memory will result in a memory allocation of a full megabyte, nearly a half meg wasted.

These two types of allocation cannot be decided on the fly, rather they are built into the system kernel. If standard allocation is desired, the standard kernel is used to boot the system. If buddy allocation is desired... you get the idea.

17.3 Memory for Processes

Prior to OS-9 Version 3.0, a process could only have a maximum of 32 different areas of memory allocated to it at a time. This does not mean that only 32 allocations could be made (without releasing memory), because if one of the memory areas owned by a process could be extended on either end, it would be. The new line of thought is that a process may have a theoretically unlimited number of different memory areas, since the memory allocation table owned by each process is no longer a fixed size. One down side to this is that now a memory allocation may take longer from time to time while the size of the process' allocation table is increased (they default to 32 entries and each time an entry is needed and none are available, another block of entries is created). But if this happened on older systems, the alternative would have been a failed memory request! Generally speaking, however, an arbitrary table size actually decreases the speed of memory allocations because the system should no longer be as concerned about extending

existing areas and assigning both areas to a single table entry.

17.3.1 malloc()

This call is one of the oldest and most common memory allocation requests in the computer world. It takes a single parameter, the number of bytes desired, and returns a pointer to at least that number of bytes, or a null pointer if an error occurs. If code is being written for portability to other systems, this call should perhaps be used. It is an efficient memory request utility that keeps track of memory allocated from the system but not yet requested by the application. This allows future requests to come from memory the application already owns but has not yet requested.

➤ When the malloc request is for, say, one byte, malloc is forced to allocate some minimum amount of memory from the system, often 4k. malloc remembers that only one byte of the 4k has been granted from this block and will grant future requests from the same block of memory as long as enough free space remains. Also, OS-9 only manages a minimum of 16 byte blocks, so when malloc gives you one byte, it really gives you 16 from the memory arena.

The function _mallocmin() takes a size as its only parameter and installs this size as the minimum number of bytes to be requested from the system if malloc should need more memory. If the size is set to a value below the minimum memory request set by the system, it will be increased to the system minimum.

The memory granted to a process by malloc is returnable by the free() routine. This memory is not necessarily returned to the system, but is returned to malloc. When malloc (through free) discovers that a large block of memory is not being used by the process, it will return that block to the system's free pool. But if the entire block is not free, the block may be used to satisfy future malloc requests.

Memory returned through free() is given back to the system in 4k increments. That value may be changed through a call to _freemin().

17.3.2 Lower Level Allocation

The subroutine _os_srqmem() is used for quicker memory allocations than malloc, with the price being memory efficiency and portability. The call takes three parameters: a pointer to the number of bytes requested, a pointer for where to store the address of the granted memory, and a color specification. The function _os9_srqmem() is similar, but works only for OS-9, while the former is portable to OS-9000.

One interesting use of this call is to request the largest block of free memory available. This will occur if the size requested is -1 bytes. The size parameter will be filled with the actual size of memory that was granted through the call. This feature is very useful if an engineer wishes to control memory allocations from within the software, rather than rely on standard system allocation methods.

If the color is specified to be a non-zero value, memory will only be granted from that area. If however, the color is specified as zero (a don't care state), memory will be searched based on priority and amount of free memory in each block until a block is found to satisfy the request.

In either case, the pointer to the memory granted and the size of memory granted with each request must be saved if the process is ever to return the memory prior to termination. Memory is returned to the system through the _os_srtmem() call which takes both size and address as parameters.

CHAPTER 18

Issues for Custom System Design

In designing a system, particularly an EPROM-based system, there are many things to be considered. This chapter explains what the kernel does at boot time to decide how the system will be handled, allowing you as a system designer to fit OS-9 and your application into a turnkey product.

18.1 Items Required to Boot OS-9

When OS-9 is started up, certain modules must be able to be found either in the system ROMs or, if applicable, on other bootable media, such as a hard drive. It goes without saying that the kernel must be found but generally many others must also be present, including the init module and any file managers / device drivers / device descriptors required for devices used immediately after the boot process.

18.1.1 Init Module

The init module is basically a data module with a particular format to it. The standard header file <module.h> contains the structure of the system init (or, configuration) module. Most of the fields in this module may be changed via the moded utility. Some of the fields contain the names of additional modules that must be present at boot time.

18.1.1.1 Number of irq Polling Entries

This is the number of entries in the IRQ polling table. One entry is required for each interrupt generating device control register.

18.1.1.2 Size of the Device Table

OS-9 devices do not have to be initialized at the time the system is booted; they may be initialized (and deinitialized) as required any time after the boot. However, the system only supports a finite number of devices at any one time, limited by the size of the system device table. This field contains the number of devices that may be supported and the table does *not* expand if it gets full.

18.1.1.3 Initial Process Table Size

OS-9 maintains a process table which holds pointers to the process descriptors for all processes on the system. The process table is basically an array of pointers to process descriptors. This array is created to be a certain size and when it is full (and a new process is created), the table is doubled in size causing a delay in process creation. The initial process table is created to hold the number of processes specified in the init module which must be a power of two.

The atomic kernel will not expand this (or any) system table. If you are planning on using the atomic kernel, your init module should declare enough space in the process table for all of the processes you plan to create.

18.1.1.4 Initial Path Table Size

The system path table is similar to the process table; re-read the previous section and substitute path descriptor for process descriptor.

18.1.1.5 First Executable Module

When the boot process completes, the OS-9 kernel needs to know where to proceed. The first executable module is the name of a program module that should be executed. On a

development system, this is often the shell or a custom program (often called sysgo) which is responsible for starting a shell on the system console. If this item is not found as a module in memory, it must be a pathlist fROM the default directory specified in one of the fields that follows. For example, "cmds/shell" would run the shell without requiring it to be previously loaded into memory.

An embedded system will usually have the name of the main application specified here.

18.1.1.6 Start-Up Parameter String

This is a text field which specifies command line arguments for the first executable module.

18.1.1.7 Default Directory Name

This is the directory the kernel should change into prior to starting the first process. Diskless systems will have a null value here. The device descriptor specified as the disk must already be in memory at boot time, and the file manager and device driver specified by the descriptor must also be in memory.

18.1.1.8 Console Terminal Name

The console terminal is used for the standard I/O path for the first process. Should the kernel encounter a system state exception, this path will be used for reporting the error. The device descriptor specified as the console must already be in memory at boot time, and the file manager and device driver specified by the descriptor must also be in memory.

18.1.1.9 Customization Module List

This field contains a list of modules which are system extensions, such as the memory management extension, SSM, or the floating point emulation extension, FPU. Each of the extension modules listed in this field are searched for, and if found, the extensions are initialized. (There is no error condition if any [or all] of the modules listed cannot

be located; any found extensions are initialized and any missing ones are ignored.)

18.1.1.10 Clock Module Name

OS-9 uses a special device driver to control the clock hardware; this driver does not conform to the descriptor / driver / manager scheme used throughout the rest of the system. This field contains the name of the driver, which must be found at boot time.

18.1.1.11 Ticks Per Time Slice

This field contains the number of ticks each process may execute, as a maximum, before being considered to a context switch. This value may be changed by modifying the system global D_TSlice after the system has booted.

18.1.1.12 CPU Type

This field is an integer defining what CPU the init module is for.

18.1.1.13 Initial System Priority

This value is the initial priority of the first process created and is usually the default for all other processes. (Of course, individual processes can be created at any priority.)

18.1.1.14 Minimum Priority

This field is used to initialize the D_MinPty global variable.The variable may be changed after the system has booted.

18.1.1.15 Maximum Age

This field is used to initialize the D_MaxAge global variable.The variable may be changed after the system has booted.

18.1.1.16 Module Directory Size

Initially the system may only hold this number of modules in memory. When more modules are required, the system will expand this table.

18.1.1.17 Initial Event Table Size

This value defines the number of events that may initially be maintained by the system. If more events than this should be needed at one time, the system will (usually) automatically expand the event table.

18.1.1.18 IOMan Module Name

The OS-9 system maintains a module which acts as the interface between the system kernel and its file managers; this field specifies the name of that module which is usually IOMan.

18.1.1.19 PreIO Module List

This text field holds a list of extension modules, similar to that of the "Customization module list," which must be initialized prior to any I/O being initialized.

18.1.1.20 System Configuration Control

This field is a word (16 bits) which defines how certain operations of the kernel will run. Bits 6 through 15 are currently undefined. The remaining fields have the following definitions:

Bit 0: if this bit is turned off, system tables will automatically be expanded when there is no room left in them. (System tables double in size when expanded, up to a maximum of 65535 entries.) If this field is turned on, tables may only be expanded through an explicit F$ExpTbl system call. (At the time of this writing, there is no C subroutine for this system function.)

Bit 1: **Currently this field is unimplemented.**

Bit 2: for the standard kernel, this bit means nothing. For the atomic kernel, if this bit is a 1, the kernel ignores the CRC value of modules during validation, thus a corrupt/patched module may be loaded fROM disk without difficulty. (This, of course, speeds up the process of module validation.) This bit is ignored for modules in ROM during the kernel cold start.

Bit 3: when turned off, the system allows system state time slicing.

Bit 4: this bit has no meaning for the atomic kernel. In the standard kernel, if this bit is 0, each user state process on the system has its own memory map for the MMU controlling software. If this bit is a one, there is only a system-wide memory map. The atomic kernel only supports a system-wide map.

Bit 5: **Currently this field is unimplemented for version 3.0.** When implemented and this bit is turned on, the SSM extension (for the MMU) will build page tables for system state.

18.1.1.21 Number of Queued Signals

Currently this field is unimplemented for version 3.0. When implemented, each process on the system will have a maximum number of queued signals rather than the limit imposed only by the amount of available memory. Once this limit is in effect, each process will still be able to expand its own signal queue through the F$ExpTbl call.

18.1.1.22 Process Descriptor Stack Size

All process descriptors have an area at the end of the structure which is the stack used when the process enters system state to perform an I/O call. This field defines how large this stack will be. Systems which have file managers and device drivers written in C will benefit the most fROM enlarged process descriptor stacks.

Thus, the items that must be found to complete the boot process are the kernel and the init module, and the modules referenced either directly or indirectly by the init module, including: the clock ticker device driver, the initial disk device descriptor and its associated driver and file manager, the main serial device descriptor (the system console) and its associated file manager and device descriptor, any system extensions that you want initialized at boot time, and IOMan or an equivalent. (Of course, if no traditional OS-9 I/O is to be performed, the device descriptors / drivers / file managers may be left out.)

Going over this list of required items, it may seem that I have left out one important item, the module which serves as the initial process run by the system. As it turns out, this module does not have to be found in memory to complete the boot. The name listed in the init module may certainly be the name of a module in memory, but it may also be a pathlist to the module on disk. If this is to be the case, the pathlist can either be relative to the disk descriptor directory specified in the init module or absolute, providing that the descriptor for the absolute path is in memory, along with its device driver and file manager.

18.2 Booting

18.2.1 Booting From Disk

To allow a disk to act as the boot device, it must be given a special file containing all of the above items listed as required to boot. This special file is nothing more than a merger of all of the other items, but it is special in that a location on sector zero of the disk must point to it so the boot software can find the file. Creation of this file and applying the pointer in sector zero is accomplished through one command, os9gen.

This command takes a list of files to merge together and creates a file called OS9Boot in the root directory of the

disk to be made bootable. Due to the number of files likely to be added to the boot file, it can be very tedious to type in this command. An example of it would appear as:

```
os9gen /h0 kernel init clock h0 rbf rbdriver term
scf scdriver ioman sysgo
```

where the disk being made bootable is the /h0 disk and everything following on the command line are the names of files to be included in the OS9Boot file.

Most systems keep the majority of the files needed in the OS9Boot file inside the directory CMDS/BOOTOBJS. Also, commonly found in this directory is a file called bootlist or some variation to bootlist. This bootlist file is usually a list of the files needed in the OS9Boot file and can be processed by os9gen as follows:

```
os9gen /h0 -z=bootlist
```

Historically, the size of an OS9Boot file could not exceed 64k. Beginning with OS-9 Version 2.4, bootfiles had pushed this limit beyond the point of being acceptable and now the file can be larger, however, os9gen still defaults to the 64k maximum. If a bootfile is to be made larger, os9gen must be made aware, as in:

```
os9gen /h0 -z=bootlist -eb=128
```

This allows the bootfile to be 128k in size.

Note: These examples make one important assumption: that the /h0 device is format enabled. The os9gen command must access sector zero to adjust the pointer to the bootfile and the RBF file manager will not allow access to that sector unless the device descriptor allows formatting. In addition, on the default device descriptors distributed by Microware for hard drives, format ability is disabled to prevent you fROM accidentally formatting your drive. However, there is a second descriptor distributed for hard drives. For the h0 descriptor, this second descriptor is called h0fmt.

As its name implies, format access is enabled for this descriptor. All other descriptor fields are identical.

18.2.2 Booting From ROM

When the system boots OS-9 from ROM / EPROM / FLASH ROM / whatever-new-technology-they-come-up-with-next, preparing the memory device is very simple. The bootstrap program knows what areas of memory may be ROM and searches for the kernel in those areas. It must find the kernel module and init module, as well as other required modules for your specific system. (See the bootlist information above for more details.)

The bootstrap also searches for any other modules that may be present, but not necessarily used as part of the boot process. As long as the init module and all other modules (as detailed above) can be found, the system will boot, and any extra modules are icing on the cake.

There is one problem associated with booting from ROM, or, for that matter, with having modules placed in ROM: EPROM access speeds are significantly slower than RAM access speeds. The latest versions of ROMBug solve that problem. In addition to the boot from ROM type options, there is a load ROM to RAM option, which transfers the modules found prior to passing control to the kernel.

APPENDIX A

The µMACS Editor

There are several different text editors available for OS-9 on the public market (and some which are commercially available through third parties). Many are similar to µMACS and many of which are different. µMACS comes from a particular family of editors all similar in usage. This appendix should help get you started with µMACS but is in no way a complete tutorial. Full proficiency with this editor requires much memorization and is gained only through reading the manual and practicing all of the possible commands.

To begin using the editor, type the command:

umacs <filename>

where <filename> is the name of a file that either exists or that you wish to create.

If you are creating a new file, your screen should be empty, except for a status bar toward the bottom. The cursor will be at the upper left of the page and, as usual, µMACS is ready for you to place text into the file.

At this point, anything you type will be entered into your file, with the exception of control characters and escape sequences, since these indicate µMACS commands. Many of the basic commands needed for µMACS are accessed via single control characters.

The μMACS editor is most often in a state of insert-mode, meaning that any text typed (with the obvious exception of command keys) is inserted into the document being edited. Table 8 details some of the most common command sequences.

TABLE 8 Common Command Sequences for the μMACS Editor

Command Sequence	Meaning
^N	Move to the **next** line in the document
^P	Move to the **previous** line in the document
^B	Move the cursor **backward** one position
^F	Move the cursor **forward** one position
^A	Move the cursor to the beginning of the current line
^E	Move the cursor to the end of the current line
^V	Move the cursor forward one page
^Z	Move the cursor backward one page
^D	Delete the character under the cursor
^H <backspace>	Delete the character to the right of the cursor
<esc> >	Move the cursor to the bottom of the document
<esc> <	Move the cursor to the top of the document
^X ^C	Exit the editor
^X S	Save the current file
<esc> Z	Save and Exit

There are certainly many other things that can be done with this editor, but it is beyond the scope of this text to teach you how to edit files. The manuals delivered with Microware's software contain a detailed set of commands for this editor, giving you a starting point if you would like to become proficient with it. The manual that comes with the operating system includes a much more extensive set of instructions.

➤ Beginning with OS-9 V3.0, µMACS has limited support for the VT-100 family of terminal emulators. That is, arrow keys will move the cursor one character or line in the expected direction. Currently, there is no support for the page-up or page-down, home, or end keys, nor any others.

APPENDIX B

Differences Between MShell and Shell

The MShell utility provides the same basic functionality as the shell utility but also enhances your productivity by providing additional commands and responses to certain keystrokes. This appendix does not teach you how to use the shell, but presents you with many of the available options.

Commands that have been entered but not yet executed (by hitting the return key) may be edited prior to execution. The line is treated as a single line file and the basic motion keys used by the µMACS editor are used to move along the line and to make changes. The control keys used for motion and control are changeable through an environment variable.

Each command that is input is placed into a history buffer of past commands. You can easily recall any command that is still in the history buffer for another execution or modification followed by execution. The size (in number of entries) of the history buffer is specified through an environment variable and the history buffer may optionally be passed from MShell to MShell and across login sessions. Previous history items may be brought to the command line and edited prior to execution.

Aliases (assigns) to commands may be installed. For example, if you are used to working in the Unix environment and

have trouble remembering that the OS-9 command to see your current (data) directory is pd rather than pwd, the command pwd can be assigned to execute pd.

You may now pipe standard error and/or standard output from one process to another.

Temporary batchfiles may be built on the command line for execution.

Filenames and previously entered commands may quickly be completed with just a few keystrokes which do not require unambiguous input.

Batchfiles may now be profiled which means they can be run inside the current shell rather than in a subshell. This allows environment variable changes and directory changes performed inside a procedure file to carry up to the shell that started the command.

The current directory can be pushed onto a directory stack and later popped off, allowing you to change your current directory (for temporary purposes) and get back to the original directory with a minimum of keystrokes.

The MShell prompt may be modified to display a number of items that may not be static, such as command history number, user id, group id, login user name, date, time and/or current data directory, among others.

MShell's largest contribution to the shell environment is its provision for a scripting language inside batchfiles. This scripting also includes the ability to expand environment variables, to contain private variables (which act as local environment variables) and to expand the output results of a command.

Symbols
.login 21, 22
_os_chain() 65
_os_fork() 64
_os_suspend() 158
_os_wait() 67
_os9_dfork() 65

A
alarm functions
    _os_alarm_atime() 92
    _os_alarm_cycle() 96
    _os_alarm_delete() 96
    _os_alarm_set() 94, 96
    _os9_alarm_atdate() 92
    _os9_alarm_atjul() 92
appending to a file 27
argc 66
argv 66

B
background operation 30

C
cc 23
chd 17
chx 24
command flags 25
command line modifiers 25
_cstart 125
cstart.r 125

D
D_MaxAge 172
D_MinPty 172
D_TSlice 172
data module functions
    _os_datmod() 77
    _os_link() 79, 80
    _os_load() 80
    _os_loadp() 80
    _os_mkmodule() 77
    _os_sercrc() 82
    _os_setcrc() 83
    _os_unlink() 82
deiniz 56
devs 56
dir 15, 16, 28, 29

E
environment 21
envtest 22
event
  _os_ev_creat() 101
  _os_ev_delete() 106

_os_ev_link() 101
_os_ev_pulse() 105
_os_ev_read() 103
_os_ev_set() 103
_os_ev_setr() 106
_os_ev_signal() 102
_os_ev_unlink() 106
_os_ev_wait() 102
_os_ev_waitr() 106
events 101
ex 37
execution directory 23

F
fixmod 47
flags 25

G
getenv() 22

H
HOME 21, 22

I
ident 47
iniz 56
interrupt
_os_firq() 147
_os9_irq() 147

K
kill 37

L
list 26, 71
load 43, 46
login 13
logout 37

M
main() 66, 125
mdir 40
memory
_mallocmin() 166
_os_srqmem() 167
_os_srtmem() 167
free() 166
malloc() 166
_mexec 125
mfree 164
moded 48, 163
module attribute
reentrant 43
state 43
sticky 43

edition 42
header 40
link count 41, 43
name 41
revision 43
Revs 42
MOTD 14
mshell 25

O

os9gen 175

P

PATH 21, 22, 47, 80, 81
path functions
_os_close() 60
_os_create() 59
_os_dup() 60
_os_gs_ready() 72
_os_open() 58
_os_read() 60, 73, 94
_os_readln() 60
_os_ss_relea() 73
_os_ss_sendsig() 72, 73
_os_write() 61
_os_writeln() 61
path descriptor 57
path_functions
_os_gs_eof() 72
pd 17
pipe 69
named 70
unnamed 69
piping 32
PORT 21
printenv 19
printf() 53
process functions
_os_chain() 64
_os_dfork() 64
_os_exec() 63, 64
_os_fork() 64
_os_setpr() 65
priority 34
stack 34
process state
supervisory state 43
system state 43
user state 43
procs 29, 30
profile 38
PROMPT 21, 22
psect 126

R
redirection of I/O 26
root directory 18

S
save 47
scanf() 53
semaphore
  _os_sema_init() 98
  _os_sema_p() 98
  _os_sema_term() 98
  _os_sema_v() 98
setenv 19, 20
setpr 38
setting the time 92
SHELL 21
shell 14, 21, 29
signals
  functions
    _os_intercept() 86
    _os_rte() 87
    _os_send() 87, 91
    _os_sigmask() 89
    _os_ss_sendsig() 90, 91
stderr 28
stdin 29
stdout 27, 28
subroutine module 123
system global
  _os_getsys() 156
system globals
  d_68881 160
  d_maxage 155
  d_minpty 158
  d_mputype 159
  d_preempt 160
  d_proc 160
  d_tcksec 159
  d_ticks 158, 159
  d_tslice 158
system() 81

T
TERM 21, 22
time slice 33
tsmon 13
types
  alarm_id 92, 94
  signal_code 92, 94

U
unsetenv 19
usage help 17
USER 21

W
w 38
wait 38
wild card characters 36

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Mark A. Heilpern has over ten years of experience in real-time programming, much of it working with Microware's OS-9 Real-Time Operating System. Before joining Microware, Mark was a technical support engineer for Ironics Incorporated.

Mark is Microware's training and education manager.

microware®

MICROWARE SYSTEMS CORPORATION

DES MOINES • TOKYO • WINCHESTER (U.K.) • AIX-EN-PROVENCE

ISBN 0-918035-04-X